Pragmatic Development of Service Based Real-Time Change Data Capture

Mitchell John Eccles

Doctor of Philosophy

Aston University

April 2012

Aston University

Pragmatic Development of Service Based Real-Time Change Data Capture

Mitchell John Eccles

Doctor of Philosophy, 2012

Thesis Summary

This thesis makes a contribution to the Change Data Capture (CDC) field by providing an empirical evaluation on the performance of CDC architectures in the context of real-time data warehousing. CDC is a mechanism for providing data warehouse architectures with fresh data from Online Transaction Processing (OLTP) databases. There are two types of CDC architectures, pull architectures and push architectures. There is exiguous data on the performance of CDC architectures in a real-time environment. Performance data is required to determine the real-time viability of the two architectures. We propose that push CDC architectures are optimal for real-time CDC. However, push CDC architectures are seldom implemented because they are highly intrusive towards existing systems and arduous to maintain.

As part of our contribution, we pragmatically develop a service based push CDC solution, which addresses the issues of intrusiveness and maintainability. Our solution uses Data Access Services (DAS) to decouple CDC logic from the applications. A requirement for the DAS is to place minimal overhead on a transaction in an OLTP environment. We synthesize DAS literature and pragmatically develop DAS that efficiently execute transactions in an OLTP environment. Essentially we develop efficient RESTful DAS, which expose Transactions As A Resource (TAAR).

We evaluate the TAAR solution and three pull CDC mechanisms in a real-time environment, using the industry recognised TPC-C benchmark. The optimal CDC mechanism in a real-time environment, will capture change data with minimal latency and will have a negligible affect on the database's transactional throughput. Capture latency is the time it takes a CDC mechanism to capture a data change that has been applied to an OLTP database. A standard definition for capture latency and how to measure it does not exist in the field. We create this definition and extend the TPC-C benchmark to make the capture latency measurement.

The results from our evaluation show that pull CDC is capable of real-time CDC at low levels of user concurrency. However, as the level of user concurrency scales upwards, pull CDC has a significant impact on the database's transaction rate, which affirms the theory that pull CDC architectures are not viable in a real-time architecture. TAAR CDC on the other hand is capable of real-time CDC, and places a minimal overhead on the transaction rate, although this performance is at the expense of CPU resources.

Keywords Real-Time Change Data Capture, Push, Pull, Data Access Services, Empirical Evaluation

Acknowledgements

I was fortunate enough to have two superb supervisors to guide me through the PhD process. I would like to start by thanking both Tony Beaumont and David Evans for their combined efforts and patience in teaching me to become a solid researcher. Without their tutelage I would not have completed this piece of research.

Furthermore, I thank Tony for giving me his time and input as a supervisor. Tony's technical knowledge and experience were an invaluable source of guidance, which helped to refine my thinking and keep me on the right track.

I thank David for his approach to supervision. David's alternative thinking style and rigorous approach to questioning prepared me in defending my work and my ideas. I also thank David for sharing his contacts and introducing me to people in industry, especially a group of researchers at IBM.

I would also like to express my gratitude to family members and friends for providing encouragement and light relief along the way. In particular, thanks go to, my mother, my grandparents, Tammy, Vik (for many great technical discussions), Nikita, Ryszard and Alina.

Finally, I would like to thank IGI for their support and understanding during the final two months of my writing up process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a brief introduction into the data warehousing field. It begins by discussing an architectural shift that is currently under way in the data warehouse field; from batch loaded static data warehouses to dynamic real-time data warehouses. A fundamental technology in the shift towards real-time data warehousing is Change Data Capture (CDC). CDC facilitates the shift from batch loaded data warehouses to real-time data warehouses, by being able to capture operational data changes more frequently than batch systems.

CDC is the focus of the study in this thesis. This chapter moves on to establish the motivation for why CDC is the chosen topic for investigation. After presenting the motivation, the thesis' aims and objectives are set out, which leads into the four contributions that are made in this thesis makes. Finally, the chapter rounds off with a thesis map, and a summary.

1.1 Data Warehousing and Business Intelligence

Data warehouses and the Business Intelligence (BI) gleaned from them are vital for organisations that are competing in today's financial market. Good BI can provide organisations with analytical information to help them understand business trends and day-to-day operations, which ultimately enables better decision making. At the heart of all data warehouse systems is the ETL (Extract, Transform and Load) system. The ETL system is used to extract data from operational databases, transform it to match data warehouse schemas and then load it into the data warehouse [81]. Traditionally, ETL systems run in batch mode, and load data into the data warehouse on a daily, weekly or monthly basis [63]. Batch ETL requires transactional systems to be put offline, because the extract process requires a lot of DBMS resources, and leaving the system online during an extraction, could affect operational transactions. Up until recent times batch ETL systems have sufficed, however, with technology improving and changes to how an organisation operates, a demand for real-time data warehousing has emerged.

Technologies such as the Internet have provided a platform to deliver information on

demand. For example, news websites can now provide up to the minute news, making newspapers somewhat out-of-date as they display yesterday's news. On demand television also means people no longer have to wait for a program to be shown according to a schedule. The demand for up-to-date information and content has now transgressed into the business world, and analysts want to know what is happening in their business now.

New technologies are required to support the demand for up-to-date information. With organisations running twenty four seven, operational systems have less opportunity to go offline to support ETL tools. Batch ETL systems are costly and inefficient because they require a lot of processing power, memory and network bandwidth in order to do their work. Furthermore, the volume of data in the operational data store is ever increasing, meaning more data has to be moved during the batch window. As the data stores grow in volume, more powerful and costly hardware will be required to run the ETL task in the available time window.

To facilitate real-time data warehousing, a new approach to ETL is required. It is no longer sufficient to extract changes in batch mode. Instead changes need to be captured as they happen in the operational systems. A technology that is capable of capturing changes as they occur is known as Change Data Capture (CDC). CDC is vital to real-time data warehouse architectures as it provides the ability to instantly capture and deliver the changes that are made to an organisation's operational data stores.

1.2 Change Data Capture

Change data capture drives real-time data warehousing by allowing one to capture fresh change data from an OnLine Transaction Processing (OLTP) database, whilst the OLTP system remain online. Database systems can remain online, because CDC can extract new change data without needing to pull the entire data set from the database. Moreover, CDC has mechanisms to identify only the newest of change data. Identifying only the latest changes alleviates the need to extract the entire data set from the database, and instead, CDC can just retrieve a small subset of change data. Selecting only the newest change data from the database is less degrading towards the database's performance and can be done whilst the database remains online. Thus CDC can obtain change data more frequently than batch ETL, which in turn allows CDC to feed data warehouses with change data more frequently.

There are two common CDC architectures; pull architectures and push architectures. In pull CDC architectures change data is found by polling a database resource. The database resource that gets polled depends on the identification mechanism at the heart of the pull CDC architecture. Typical pull CDC identification mechanisms include: tagging the operational data with timestamps, where change data can be found in the operational tables; using triggers to place fresh change data into a staging table, where change data can be found in the staging tables; enabling database redo logs, where change data can be found in the database redo logs.

High transactional performance is critical in an OLTP system [81], and a limitation of pull CDC architectures is the DBMS resource queries, which may place a significant overhead on the database's transaction rate. The polling operation that is integral to pull CDC techniques could place a performance strain on the OLTP database if it is executed too frequently. A frequent polling query could conflict with the OLTP system's high transactional throughput. The polling queries will compete with the transactions for the database's resources. The more frequently the poll operation is executed, the more competition there will be for the database's resources, which will eventually lead to operations being blocked, and the database's transaction throughput will be reduced, as transactions will take longer to complete. To minimise the impact pull CDC could have on the database's transactional throughput, an interval can be placed between successive poll operations. However, such an interval delays the extraction of data changes, which limits pull CDC's real-time potential. The longer the poll interval, the more time can elapse before the change data can be used in real-time analysis, which means the change data potentially loses value. Therefore, it is unlikely that pull CDC architectures are viable in a real-time architecture.

Push CDC architectures on the other hand capture changes before they are sent to the database. Capturing data changes before they are sent to the database can be done in the application that is making the change. Placing the CDC logic inside the data modification applications removes the need to poll the database. Instead change data can be captured from the memory context of the application that has made the change.

Theoretically, push CDC architectures will have less latency than pull CDC architectures, because they do not compete with the transactional workload for database resources in order to obtain changes, and thus it does not require polling and a poll interval. However, push CDC architectures are seldom implemented, because organisations have many data modification applications, and placing CDC code in each application would lead to an arduous maintenance task [8]. Additionally, application based CDC will be highly intrusive towards existing applications that do not have CDC logic, as these applications have to be modified and recompiled to support CDC. To make push CDC architectures viable and to capitalise on push CDC's potential as a true real-time CDC technology, a more viable implementation is required. A viable approach to push CDC will be easier to maintain, and less intrusive towards system applications.

1.3 Thesis Motivation

The motivation for the study in this thesis is to provide a viable approach for building push CDC architectures. In our extensive literature searches, we have found little evidence that a viable push CDC mechanism is being developed. Furthermore, we have found no empirical data on how much latency there is in capturing data changes with the push model. A viable solution to push CDC will potentially provide a boon to the real-time data warehousing field, because it will provide system architects with a low latency alternative

to the current pull models.

To verify that the push CDC architecture does indeed have less latency than pull CDC architectures, we shall require an empirical evaluation of our push CDC architecture. An empirical evaluation of our push CDC architecture will allow us to quantify the advantages of using a push CDC mechanism over a pull CDC mechanism. A secondary motivation for an empirical study on CDC architectures comes from the lack of published data on the latency that is caused by pull CDC architectures. Few empirical studies have been conducted to underpin the theory of resource competition in pull CDC architectures; despite practitioners suggesting that pull CDC can do real-time CDC. It would be advantageous for the field to have empirical data to determine: the impact polling has on transactional throughput; the cost pull CDC has in terms of system resource usage; how much the polling interval delays the capture of fresh data; the feasibility of pull CDC in a real-time environment.

1.4 Thesis Aims and Objectives

The previous section provided an overview of the motivation for the work in this thesis. Considering the motivation, the aim of this thesis is to:

- pragmatically develop a push CDC architecture and empirically determine its viability as a real-time capture technology, by evaluating it against pull CDC architectures.

To achieve these aims; the following research objectives can be defined:

Objective 1 Design a push CDC architecture that will address the software engineering difficulties that exist in current push CDC architectures.

Objective 2 The proposed push CDC approach will not prevent a high transactional throughput on an OLTP database.

Objective 3 Design an experiment to benchmark the performance of CDC technologies.

Objective 4 Use the results obtained from the CDC benchmarks, to evaluate our push CDC architecture, by contrasting it to the performance of pull CDC architectures.

1.5 Thesis Contributions

The contribution of this thesis is realised by attaining the four objectives set out in the previous section. This section considers how attaining the thesis objectives will lead to contributions in the data warehousing field.

A proposal for a viable push CDC implementation. We make a theoretical contribution to the CDC field by addressing the inadequacies of current push CDC implementations. We propose a service based CDC methodology, which involves encapsulating the database with a set of DAS. The DAS will decouple data access and data modification code from the application, which in turn will allow us to decouple CDC logic from the application, thus providing a robust and maintainable push CDC architecture.

A pragmatic evaluation of DAS in an OLTP environment. We synthesize DAS literature to create a methodology for DAS that is viable in an OLTP environment. We affirm our discussion with an empirical evaluation of DAS methodologies and implementations to find the optimal DAS design for an OLTP environment. Previous works in the literature have evaluated DAS; however these DAS were evaluated for performance in database environments that have different requirements to OLTP databases. We then further evaluate the chosen DAS design with the industry recognised TPC-C benchmark [26]. The TPC-C benchmark simulates a simplified OLTP production system, so that one can gain insight into the performance of a database server's hardware and software configuration. For the purposes of our study, the TPC-C benchmark is used to determine the cost of conducting OLTP transactions via a set of data access services.

Design of an experiment to evaluate CDC. We design an experiment based around the TPC-C benchmark to measure key CDC performance metrics: impact on transaction rate; amount of capture latency; impact on CPU usage. The TPC-C benchmark is ideal for providing an insight into the impact CDC has on the database transaction rate. However, the field does not have standardised approach for measuring capture latency. Measuring capture latency is not trivial. To take a time measurement, one requires two points of reference. Deciding upon the position of two points of reference is crucial to making an accurate measurement. If the reference points are in the wrong place, then capture latency measurements could become inconsistent. We discuss potential possibilities of where to take the points of reference, before specifying a definitive technique for measuring the capture latency in a given CDC architecture. The two reference points are: 1) the time the record gets committed to the database; 2) the time the record is fully captured from the database. Precisely measuring the time, at which a change is committed to a database, is not a feature commercial databases offer. We had to devise a novel system for recording commit time. We successfully extended an implementation of the TPC-C benchmark to use our measurement process. Theoretically, our CDC latency measuring mechanism could be used in any OLTP system, and not just the TPC-C benchmark. Ultimately we provide the field with a method for measuring capture latency in CDC architectures.

Interpretation of empirical data. A major contribution of this thesis is the collection, analysis and interpretation of empirical data on the performance of CDC technologies. An exiguous amount of empirical data has been collected and published on the performance of CDC technologies. Our experiments measure CDC performance in a range of scenarios. Our interpretation and analysis of the collected data helps to underpin

the theory behind CDC technologies, and to empirically determine the latency that they cause. This contribution also opens up the discussion on CDC capture latency and provides a platform, which other researchers can use to extend and deepen the understanding of latency in real-time data warehouse architectures. The results and analysis of these experiments will also be useful to practitioners who can incorporate the data into their decision making process, when deciding to build a real-time data warehouse architecture.

1.6 Thesis Organisation

The rest of this thesis is organised as follows. Chapter 2 provides background information of the related work so that: 1) an overview of the field is provided; 2) the context of the work in this thesis is defined; 3) the need for empirical data on CDC latency is highlighted; 4) the need for a viable push CDC solution is highlighted. Chapter 3 introduces a technique for addressing the challenges of push based CDC technologies. We devise a technique, which involves doing CDC through a set of data access services that encapsulate the transactional database. Chapter 4 presents an investigation into the most efficient approach to develop DAS for OLTP environments. We discuss competing DAS methodologies and implementation techniques, before providing a preliminary evaluation of the various DAS approaches. The aim is to find the implementation that has the least impact on the transaction rate. Chapter 5 extends the preliminary experiment from Chapter 4, by benchmarking the transaction rate of the prevailing DAS methodology, using the industry accepted TPC-C benchmark. Chapter 6 describes how we setup an experiment to evaluate the performance of CDC technologies. This includes describing our novel approach to measuring capture latency. Chapter 7 presents the results obtained from the CDC evaluation. Chapter 8 analyses the performance of the CDC technologies by interpreting the meaning of the results. Chapter 9 concludes this thesis, and outlines possible directions for future work.

1.7 Summary

This chapter set a foundation for this thesis, by providing the background and motivation for a study that focuses on change data capture architectures. We identified two distinct CDC architectures; push and pull architectures, and briefly debated over their viability within real-time data warehouse architectures. With the motivation in place, we identified the aim of this thesis, and set out four objectives that we shall need to attain in order to achieve our aim. Attaining the four objectives will subsequently allow us to make four contributions to the data warehousing field.

Chapter 2

Change Data Capture

This chapter expands on the introduction to the thesis, by providing an in-depth analysis of the change data capture field. The chapter begins at a high level, by discussing the need for real-time data warehousing, and importantly, sets out the definition of real-time for this thesis. The chapter then moves on to provide an overview of how fields that are related to CDC, are working towards the primary goal of real-time data warehousing; which is to reduce the time it takes to populate the data warehouse with fresh data. The remainder of the chapter then focuses in on the CDC field. The CDC discussion provides an in detail examination of CDC architectures. This discussion incorporates a review of the two existing architectures and the CDC mechanisms that are associated with them. The discussion highlights the flaws with both architectures and points out the need for empirical data to objectively underpin these flaws. This chapter concludes by reflecting on the current state of the CDC field, and proposes a CDC mechanism, which has potential to overcome the flaws associated with its predecessors. The chapter also sets out the aim to empirically evaluate our proposed CDC mechanism against existing solutions.

2.1 Real-Time Data Warehousing

It has been over twenty years since Inmon [54], established the concepts of data warehousing, and the benefits of data warehousing are now well understood. In the early days of data warehousing it was fine to supply the data warehouse with fresh data on a daily or weekly basis. However, with the uptake of the Internet, and an insatiable need for up-to-date information, the real-time data warehousing field has prevailed [102].

The primary goal of real-time data warehousing is to reduce the time it takes to load the data warehouse with fresh operational data, and ultimately minimise the time between a business event taking place and analysis being performed on that event [13], [59]. A selection of surveys and empirical studies by Wang and Strong [108], Shin [100], Mannino and Walter [72] suggest that the freshness of data in the data warehouse is key to the success of the information system. Furthermore, industry experts and researchers Jörg and Dessloch [59], Bruckner et al. [13], Reimers [95], Cochinwala and Panagos [24], Bhensda-

dia and Kosta [9], recognise that real-time data warehousing is vital for timely business decision making, which can prevent organizations from falling behind their competitors. Reimers [95] gives an example where one organization noted an 18% increase in hourly sales after implementing a real-time data warehousing system. Real-time data warehousing can also be used to prevent fraud. The telecommunications industry can use real-time data from data warehouses to quickly detect and subsequently prevent fraudulent activity on their network [24], [78]. Airline and government agencies also benefit from the ability to analyse operational data in real-time, as it allows them to quickly detect suspicious groups of passengers or potentially illegal activity [66].

The benefits and the need for real-time data warehousing are well established. The faster the operational data can be captured and analysed the more money can be saved. Hackathorn [48] depicts the value of real-time data warehousing with a succinct graph, shown in Figure 2.1. Hackathorn's graph demonstrates that there are three types of latency that delay decision making: 1) capture latency, the time to capture fresh data; 2) analysis latency, the time to analyse the captured data; 3) decision latency, the time to make a business decision based on the analysis. If the summation of these three latencies is too great, then the business value of the operational data diminishes, and opportunities can be missed. The goal in the real-time data warehousing field is to further reduce these three latencies and increase the business window of opportunity.

Figure 2.1: A Graph Depicting the Value of a Business Decision vs Time, Taken From Hackathorn [48]

Before we discuss how researchers in the field are working to further reduce latency in data warehouse systems, we define what we mean by real-time. The term real-time gets blurred in some of the literature, and the following section looks at the meaning of real-time and sets to clarify the use of the term, for the purpose of this thesis.

2.1.1 The Meaning of Real-Time

It is important to understand what is meant by real-time because it has several different interpretations when used to describe real-time data warehousing. The Data Warehousing Institute (TDWI) conducted a survey, with 423 respondents, to get a better understanding of the best practices in the data warehousing field [34]. One of the questions in Eckerson's survey was aimed at discovering the term respondents use to define real-time business intelligence. The results from the question, shown in Figure 2.2, highlight the ambiguity over the definition of the term real-time; there are several phrases which describe a similar thing. Some of the meanings or references to real-time are used interchangeably, which causes confusion over how real-time is defined [95]. The purpose of this section is to clarify what is meant by real-time in terms of this thesis. We also provide an argument for why some of the other terms, in particular near real-time BI and right-time analysis are distinguishable from real-time.

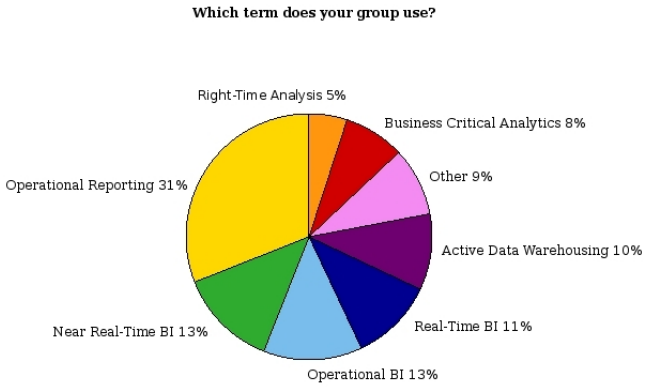


Figure 2.2: TDWI Survey Based on 423 Respondents: Which real-time term does your group use?

The confusion around the definition of real-time stems from the lack of a strict, universal agreement on how much time can pass before target applications are supplied with fresh operational data. For example organisation A's target applications may require delivery of fresh data changes within ten minutes of the data being created, whilst organisation B's target applications, may require delivery of fresh data changes within one hour of the data being created. Despite the differing latencies both organisation A and organisation B may consider their systems to be real-time, so long as the fresh data is delivered to the target applications within the given deadline.

The TDWI [34], label operational BI and real-time BI, as systems that delivers information to a broad range of users within hours or minutes. Other practitioners and researchers use the term near real-time to describe a similar latency. Reimers [95], interviews a data analyst who considers near real-time to be a system that collects fresh data within an hour or in some cases within twenty four hours. The same analyst considers a "strict" real-time system as a system that collects fresh data within less than a minute. Terr [102] considers a system that has negligible latency as real-time, and a system with

a tangible time frame such as two hours as near-real time. Cochinwala and Panagos [24] presents a near real-time business intelligence solution for telecommunication data. Their definition of near real-time is to collect fresh data within five minutes.

One can see that there is not a universal definition to distinguish between latency in a real-time system and latency in a near real-time system. Cochinwala and Panagos [24] considers five minutes to be near real-time. Whereas Reimers [95] suggest that data collection within a minute is real-time, and agree with Terr [102], that data collection within an hour or hours is near real-time. The natural question to ask is: *When does a system become real-time rather than near real-time?* Terr's [102] idea that the latency in a near real-time system is tangible whilst the latency in a real-time system is negligible is not strong enough to distinguish between the two. *What time does one consider as negligible?* Theoretically, a millisecond is tangible and not negligible, when the target application needs the data in a microsecond. To distinguish between real-time and near real-time we propose that one must consider the concept behind the two systems, rather than separate them with a time frame.

Near real-time systems provide targets with fresh data right when they need it, rather than providing them with fresh data as soon as it is available. In this sense, right in time or just in time are also applicable terms for this type of system. In fact some sources do define this type of system as a just in time system [42]. Essentially, near real-time is an implementation of Kimball and Caserta's [63] microbatch ETL, where ETL batch windows run more frequently, say on a minutely or hourly basis, rather than daily or weekly. Each batch window will run just in time to extract fresh data from the source and transform it, before sending it to the target. The concept behind near real-time is to only extract fresh data, from the source, when the target needs it. Near real-time systems have a built in delay, which means changes are only captured when they are needed. We define the built in delay as *mechanical latency* in Definition 1. This mechanical delay also makes near real-time more affordable than a real-time solution. Jörg and Dessloch [59] and Langseth [66] consider near real-time to be more feasible and less expensive, because one will not require high performance hardware and software, to capture fresh data changes immediately. The TDWI also recognise, that a near real-time approach is the first step towards a real-time architecture [34].

Definition 1. ***Mechanical Delay****: is a delay that has been introduced into the system by the system's designer.*

The concept behind a true real-time system is to get fresh data to the target without any mechanical delay. To achieve the aims of true real-time, an extract process will have to continuously extract fresh data from the source. The continuous extraction of fresh data creates a data stream from the source to the target. It is this continuous stream of data, without mechanical delay that makes a real-time system [42, 66]. Without the continuous stream, one ends up with the aforementioned microbatch ETL concept, and thus a near-real time approach. Definition 2 is the definition of *real-time* that will be used

in this thesis

Definition 2. ***Real-time****: is where a system moves fresh data from a source to a target without mechanical delay.*

Of course, some may still challenge definition 2. Bruckner et al. [13] suggests that delays in capturing changes and propagating them across the network, will invariably mean that there is latency in the data warehouse architecture. Therefore, a system can only ever be near real-time, as zero latency is never physically possible, and thus they use real-time and near real-time interchangeably in their works. We agree with Bruckner et al. that there will invariably be some delay, but we define this as *processing delay* in Definition 3. A real-time system will naturally have processing delay, as the nature of time means no process can be complete in less than or equal to zero time.

Definition 3. ***Processing Delay****: the delay that naturally occurs when a computer system is processing instructions.*

Finally we clarify the meaning of near real-time in Definition 4. Near real-time can be used interchangeably with just in time or right in time. A near real-time system can have both mechanical and processing delay, so long as it delivers fresh data to the target within the target's deadline.

Definition 4. ***Near Real-time****: is where a system delivers fresh data changes to a target, in a deadline that meets the target application's deadline.*

Definitions 1, 3, 2 and 4 are used throughout this thesis to evaluate the real-time viability of change data capture mechanisms.

2.1.2 Research on Real-Time Data Warehouse Architectures

Real-time data warehousing begins with capturing operational data from data sources in real-time. Real-time data extraction is considered as a critical component of the real-time data warehouse architecture [13, 47, 88]. Without real-time data extraction, there will be no data stream, and thus no real-time data warehousing.

The prevailing technology for real-time data extraction is Change Data Capture (CDC). CDC was introduced in Chapter 1, and a more comprehensive discussion on CDC is presented in Section 2.2. Before real-time data capture is discussed in depth, it is worth considering other components in the real-time data warehouse architecture, to demonstrate some of the problems other researchers are attempting to solve. This is done by considering: 1) how changes are propagated through a real-time data warehouse architecture in Section 2.1.2.1; 2) how changes are processed in a real-time data warehouse architecture, in Section 2.1.2.2; 3) how changes are loaded into a data warehouse in a real-time data warehouse architecture, in Section 2.1.2.3.

2.1.2.1 Real-Time Change Propagation

Once the operational data has been captured from the data sources, one must transport the data to disparate target systems, which will utilise the change data. There are two prevailing trends for propagating data between the extraction tools and the users of the change data: 1) publish subscribe mechanisms, where the capture tool will push change data to subscribed targets; 2) Enterprise Services Buses (ESB), where capture tools and targets share a common data bus for communication purposes.

One method for transferring data between processes could be to have the targets poll the CDC tool, by periodically making requests for new data changes. Figure 2.3 shows an architecture where targets pull data from the CDC tool in a request/response workflow. Target systems such as an Operational Data Store (ODS), a transformation and loading tool (T) or an Event Analysis Engine (EAE), will request new changes from the CDC tool at regular intervals. The CDC tool will respond by passing on new change data if it has it available. Pull architectures are not conducive to a real-time architecture, as they do not create a continuous flow of data. Furthermore, it seen as an inefficient model for populating data warehouse systems [67].

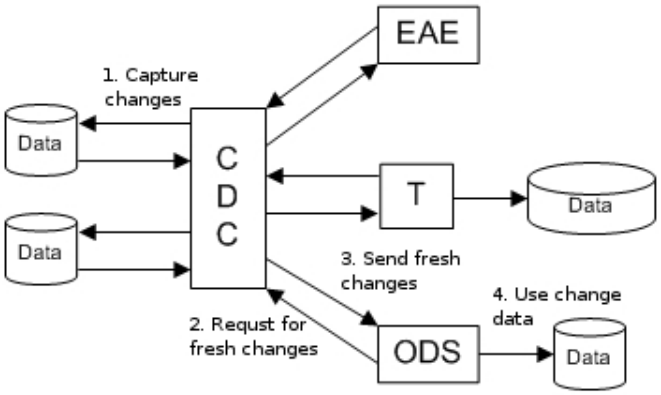


Figure 2.3: An Architecture Where Targets Tools Pull Change Data From a Capture Tool

Researchers started to look for alternatives to the pull architecture. The alternative to pull architectures is push architectures. In push architectures the data capture tools will push data towards the target tools.

A popular push model is based on the publish/subscribe pattern which is derived from Gamma et al.'s ("Gang of Four") [43] observer pattern. A publish/subscribe system will allow the capture tool to propagate change data to distributed target applications in real-time [70]. Figure 2.4 shows how the data warehouse architecture from Figure 2.3 can benefit from a publish/subscribe system. In this architecture, the targets will first register their interest with the CDC tool, and signal that they wish to receive change data. When the CDC tool obtains fresh data changes from the operational databases, it will send copies of the data to registered subscribers. Now the targets will automatically receive change data when it is available, rather than having to request it from the source. This creates a

more natural flow of data.

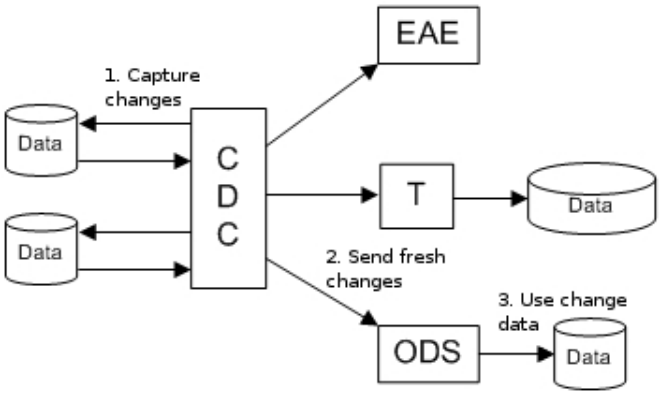


Figure 2.4: An Architecture Where the CDC Tool and Target Communicate via a Publish Subscribe System

Another push-based architecture is based around message buses, known as Enterprise Service Buses. An ESB provides a convenient way to decouple the target application from the data extraction tools. An example of an ESB being deployed in a real-time data warehouse architecture can be found in [78]. Figure 2.5 shows how the data warehouse architecture from Figure 2.4 can utilise an ESB. When the CDC tool captures fresh change data, it can wrap the change data up as a message and place it on to the message bus. The ESB then propagates the message to targets that are connected to the bus. The CDC tool will no longer need to know about the subscribers or where it needs to route the change data to, other than that it needs to place a message on to the ESB.

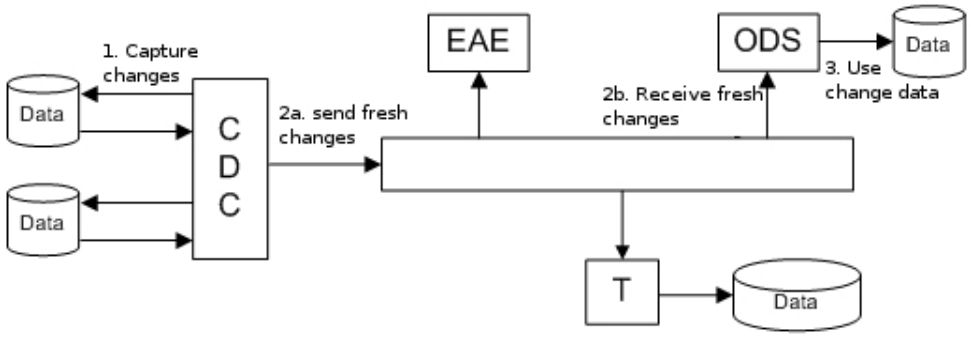


Figure 2.5: An Architecture Where the CDC Tool and Target Communicate via an ESB

Ultimately, a push-based delivery mechanism is seen as an efficient mechanism for propagating change data to target systems in real-time data warehouse architectures.

2.1.2.2 Real-Time Change Processing

Processing changes in a real-time system poses a different paradigm to processing batches of changes in traditional ETL. Real-time is all about minimising latency; once changes have been captured, they are immediately propagated to processing tools. The immediate

propagation of changes results in a stream of changes. Processing tools process this stream to: clean the data; transforming change data [18, 30, 90]; do immediate analysis on the stream, such as in an event driven architecture [78]. The stream of changes can appear to be infinite, and once a data change arrives at a processing tool, it is best for the tool to process it immediately [79].

Vassiliadis and Simitsis [105] recognise that the biggest issue in real-time stream processing is the optimisation of system resources. Moreover, Dayal et al. [28] see the problem as trying to optimise freshness, reliability, scalability, maintainability, recoverability and fault-tolerance. To combat the stream processors need for system resources Thomsen et al. [104] and Polyzotis et al. [90] look to better utilise memory resources. In main though researchers have turned to distributed computing [2, 21, 79] and grid computing [79, 17] to solve the computational requirements of stream processing. Nguyen et al. [79] suggest that the parallelism of grid computing is powerful enough to handle the significant amounts of data that are generated in a real-time data warehouse system.

2.1.2.3 Real-Time Change Loading

Data warehouses exist to provide a static, historical picture of an organisation's business operations, and supports complex, ad-hoc queries, so that analysts can gain business insights [19]. Data warehouses are designed to efficiently execute analytical queries, thus the data warehouse DBMS is tuned for data reading, rather than data writing; the reverse of an OLTP database. A clear distinction between querying and updating is a fundamental concept of data warehousing [90]. Continuously updating data warehouse tables contradicts the purpose of having a specialised read-only database, and performance will be heavily compromised [66]. Therefore, the real-time stream of fresh data places a burden on data warehouse resources, which has lead researchers to find alternative ways to load the data warehouse.

A prominent approach to supporting the real-time data is to create a real-time data partition in the data warehouse. Kimball, a data warehouse guru, advocates the use of a real-time partition in the data warehouse. A real-time partition is a separate set of tables from the static data warehouse. The real-time partition should contain all of the activity since the last update on the static data warehouse, link seamlessly to the static data warehouse, support high performance querying [62]. The real-time partition is a light-weight version of the real data warehouse, without any aggregation. The real-time partition exists to enrich the data in the static warehouse, and the data is eventually copied over to the static data warehouse tables at a time that is convenient to the DBMS server's resources [62].

An alternative solution to allow the data warehouse to cope with a continuous stream of data, is a shift in data warehousing paradigm, by moving towards active data warehousing [60], also known as real-time data warehousing [14]. An active data warehouse is a data warehouse that is specifically designed to remain online whilst data is being loaded into it. Karakasidis et al. [60] propose that the loading of active data warehouses can be tuned

with a queue based system, which essentially acts like a buffer and drip feeds the data warehouse with fresh data.

Finally, some OLAP features have been extended to directly support the data streams. A common feature of a data warehouse is multi-dimensional analysis, Chen et al. [20] propose an approach for online building and processing of multi-dimensional OLAP cubes.

2.2 Change Data Capture

Change data capture (CDC) is commended by Zhu et al. [111], Wang and Liu [107] and Pareek [88], as a technology that enables real-time data warehousing. CDC is a process that is used to capture the data changes that are applied to a particular source database; most often an OLTP database [80]. Moreover, CDC only captures changes that have not been previously extracted [8]. This is in contrast to traditional ETL systems, which would extract all of the database's data in batch either nightly or weekly [53]. In ETL systems, changes were discovered, by computing change deltas, which are the difference between the current data set, and the previously extracted set. This was a costly process that required systems to be put offline. Because CDC extracts only the newest changes, it can run alongside the database. This also means that change data can be extracted on a more regular basis, which is why it is seen as a real-time enabler.

As well as real-time data capture, the best CDC mechanisms will be able to:

- Identify the changes made to the data by the execution of Data Manipulation Language (DML) statements
- Identify the changes made to the data by the execution of Data Definition Language (DDL) statements
- Capture data changes for a variety of DBMS vendors
- Limit the impact on the transactional throughput, this aim is particularly for OLTP systems.
- In an OLTP environment it is advantageous for CDC tools to maintain transaction consistency when capturing changes. Moreover, it is useful to capture, and group the DML changes that were made in a single transaction
- Propagate changes to the target systems.

In the following sections we discuss CDC in depth, and present the various architectures and technologies that enable CDC. The purpose of this discussion is to: demonstrate the state of the art; expose areas that require more knowledge; discover where to reduce latency in CDC. Our review leads us to argue that the current mechanisms for doing CDC require mechanical latency (Definition 1), which prevents them from being real-time enablers in accordance to Definition 2. Additionally, we demonstrate that alternative CDC

mechanisms could enable real-time CDC, if they were not intrinsically flawed due to their implementation.

2.3 Change Data Capture Architectures

There are a number of mechanisms one can use to do change data capture. Each of these mechanisms imposes one of two architectural styles on the overall CDC system. The two architectures are: 1) Pull CDC, where a CDC mechanism requests change data from the database; 2) Push CDC, where the CDC mechanism detects change data on its route to the database. CDC mechanisms that impose pull architectures exist in abundance, due to the relative ease in which they can be implemented. Push CDC mechanisms are seldom implemented, but have the advantage of being better poised to enable real-time data capture. In the following sections we discuss pull and push CDC architectures, and review implementations of the CDC mechanisms that impose these architectures.

2.3.1 Pull Change Data Capture

Pull change data capture is a CDC architecture in which the capture mechanism extracts data changes from database tables or some other database resource. In other words, the CDC mechanism has to query a database resource to extract fresh changes.

A pull CDC mechanism uses three sequential processes to find new changes: 1) change identification; 2) change capture; 3) change delivery. The following sections give an insight into how each of these processes work. After this, in Section 2.3.2, we review technologies that enable pull CDC. Then in Section 2.3.3 we review the pull CDC architecture and consider its viability as a real-time CDC solution.

2.3.1.1 Change Identification

Change identification is a change detection technique that flags new Data Manipulation Language (DML) statements. Having a mechanism to identify the changes before they are extracted is the reason why CDC is an improvement over the batch extraction process. In CDC, the capture process now only has to retrieve the newly flagged changes, and not the entire data set. There are three commonly used techniques to detect whether a change has been made to the source data: 1) log-scanning; 2) triggers; 3) timestamping records [33]. These three techniques are explained individually in Section 2.3.2, but essentially they work by flagging data changes, which makes them accessible for the capture process. Moreover, DML changes can be identified by recoding the DML operations in a database redo log, or by moving the changed data to a staging table, or by adding a timestamp to the changed row.

2.3.1.2 Change Capture

The capture process is subsequent to the identification process. The purpose of the capture process is to extract the newly identified changes. Depending on the identification technique used, the capture process will find new changes in either the redo log, a staging area, or in the operational tables. The capture process will poll the necessary resource, to find the latest changes. Each successive poll operation on the given resource will allow the capture tool to retrieve the latest set of changes, i.e. the changes that have occurred since the previous poll operation.

To ensure that only the most recently identified changes are extracted, the capture process requires its own diligent identification logic. Such logic will allow the capture tool to find only the changes that it has not previously extracted. Without such logic, the extraction process will invariably extract all of the change data from the change resource. Extracting all of the data changes would negate the purpose of CDC, because as time goes by, the volume of data in the change resource will grow and extracting all of it will be no different to the previous batch extraction technique. The capture identification mechanism is dependent on the change identification mechanism that is used. When timestamp identification is used, the capture process will select rows where the value in the timestamp column is greater than the highest timestamp value of the records from the previous extraction. When trigger CDC is used, the capture process can use primary key values on the staging area to select the latest changes. This is done by selecting records where the primary key is greater than the highest primary key value from the set of records retrieved from the previous extraction. When log scanning is used, the capture process will have to keep track of its position in the log file, so that it only pulls changes that occur after its current position.

2.3.1.3 Change Delivery

Change delivery is a final process that can be included in CDC systems[1]. The purpose of delivering changes to targets is so that the targets can process the change data in a way that is beneficial to an organisation's business operations. Target systems can be hard programmed into the capture tool. Moreover, the address of the target applications could be written into the CDC code. Alternatively, in Section 2.1.2.1 we discussed how the publish/subscribe pattern and ESBs act as a transport mechanism for a continuous data flow. The publish subscribe/pattern and ESBs can be used for change delivery. Using the publish subscribe method, the capture tool will send data changes to all subscribers. With the ESB approach, the ESB will be the capture tool's target, and the ESB will allow the change messages to be propagated to the ESB's targets.

[1]Not all CDC systems need a delivery process. This is especially the case when the capture process is built into the target system

2.3.2 Pull Change Data Capture Techniques

There are a variety of techniques one can use to identify the data changes that have been made to the source database. The following sections review the state of the art in this field, and discuss the advantages and disadvantages of each technique.

2.3.2.1 Log Scanning Change Data Capture

Scanning database redo logs is the prevalent approach to doing change identification. There are two ways in which this method of CDC can be enabled: 1) DBMS log scanning, where the DBMS product has a process for scanning the redo log to find changes; 2) external log scanning, where a tool that is external to the DBMS scans the log to find changes. Section 2.3.2.2 and Section 2.3.2.3 discusses these two approaches respectively.

2.3.2.2 DBMS Log Scanning

DBMS log scanners are processes that are spawned within the DBMS to coordinate the reading of redo logs for the purpose of CDC. Microsoft's SQL Server [73] and Oracle's DBMS [65] both have features that are capable of doing log based CDC. Figure 2.6 depicts a pull CDC architecture, where DBMS log CDC is the driving mechanism for change identification. A process within the DBMS will periodically scan the redo log, seeking out DML statements that have been applied to the database. The DBMS scanner will then effectively reverse engineer the DML statement, and insert a change record in to a staging area. The change record contains the change data, and will often include additional meta-data about the change, such as the change operation (INSERT, UPDATE, or DELETE), the time the change was made, and the user who made the change. A capture tool then periodically polls the staging table to retrieve the freshest changes, before propagating them to the targets.

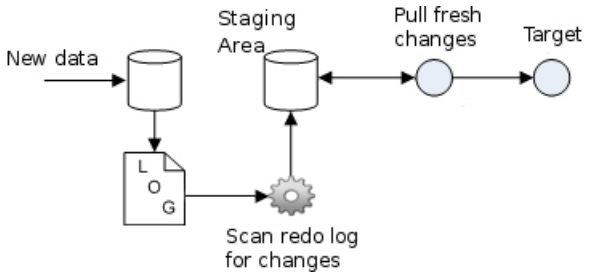


Figure 2.6: Architectual Overview of Internal Log CDC

The log CDC implementation in SQL Server, works by periodically scanning the SQL Server transaction log. As new changes are detected in the log, the change data is copied to specialized change tables. The change tables are replicas of the source tables, with an additional 5 columns for storing meta-data relating to the recorded change. CDC is enabled and disabled through the use of stored procedures. The enabling stored procedure creates two SQL Server Agent jobs: one to capture data changes; one to clean up the

change tables, which keeps change data to a minimum. The agent that scans the log is setup with default parameters to process a maximum of one thousand transactions per scan cycle and wait five second between cycles [73]. Despite the ability to change these parameters, the server agent cannot be set up in a way eliminates the latency between log scans. This latency is essentially a mechanical latency that is built in to the process to prevent the redo log from being overwhelmed with read requests. Typically, all data changes are written to a redo log for durability, in the event of system failures. If the CDC process were to be continuously reading the redo log, then write requests may get blocked, which would mean transactions take longer to complete. High transactional throughput is a priority in an OLTP system [81], so mechanical latency is introduced to maintain a high transaction rate. Therefore, it is likely that Microsoft's approach cannot be used for real-time CDC. However, Microsoft does not claim that their approach can be used for real-time CDC.

Oracle's DBMS log CDC is built around Oracle Streams. Oracle streams are a mechanism to either move data between tables on the same Oracle database instance, or to a tables on a remote Oracle instance [94].

To set up DBMS log CDC in Oracle, one must identify the tables of interest and add these tables to a change set. A change set logically groups change data [81]. A change set contains one or more change tables. Each change table corresponds to a source table. As committed DML statements are written to the redo log, an internal process will be scanning the redo logs for newly committed DML statements. When a committed DML statement gets written to the log, the internal process publishes the change data in the relevant change table, where the capture tool can then retrieve it via a query.

An advanced feature of Oracle's DBMS log CDC, compared to Microsoft's, is the ability to identify transactionally consistent changes. Moreover, this is the ability to identify the DML operations that formed part of a single transaction. As part of the metadata in the change table, Oracle stores a Transaction Identifier and a Commit System Change Number (CSCN). The transaction identifier, indicates which transaction the change belongs to [80]. The CSCN also indicates the sequential order in which the transaction executed the DML statements [80]. The fact that the change set is used to logically group transactional change data means that the target will be able to reconstruct whole transactions and not just single DML statements. This makes the capture data more valuable.

The pitfalls of both Oracle's and Microsoft's respective DBMS log CDC is the inability to propagate changes to targets that are outside of the database. Although these techniques can be quick to identify changes, they can prohibit real-time analysis, due to the fact that one must first retrieve the changes from the database in order to use them. However, it is the pull action that is responsible for excessive latency, we discuss why this in greater detail in Section 2.3.3.

Another deterrent to DBMS log CDC is the fact that log scanning process will use up DBMS resources, such as CPU and memory. Pareek [88] show how extra CPU usage, and additional overheads on the redo log, have a negative impact on an OLTP database's

transactional throughput. In OLTP environments, it is paramount to maintain high transaction rate, so that the DBMS does not become a bottleneck in business transactions. One might be able to increase CPU power, to circumvent this problem. However, doing so is reacting to a problem, rather than preventing it from occurring again in the future, as the business grows. Ideally a better CDC solution will not rely on hardware upgrading.

A final pitfall of DBMS log CDC is that it is not a standard feature of a DBMS. Unless one has either the Oracle enterprise edition or SQL Server, one cannot benefit from this type of CDC. For example, an organisation running a MySQL DBMS would have to consider switching to one of the aforementioned products at a cost, in order to benefit from this type of CDC. Conversely, those already using SQL Server, or Oracle, will have DBMS log CDC available to them.

2.3.2.3 External Log Scanning

External log CDC, is similar to DBMS log CDC, in the sense that the database's redo log is used to identify the changes that have been made to the database. The major difference is that a tool that is external to the DBMS will coordinate the reading of the redo logs. Figure 2.7 depicts a pull CDC architecture where an external log scanner is the driving capture mechanism. The external log scanner will periodically scan the redo log, seeking out fresh DML statements. The log scanner will then reverse engineer the DML statements to obtain the change data, which then gets sent on to the target.

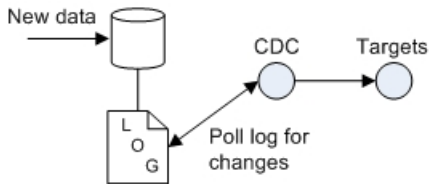


Figure 2.7: Architectual Overview of External Log CDC

External log CDC is a popular approach to doing CDC [99, 85, 91, 50, 52, 109, 51, 8, 53]. The reason for its popularity is because it can be retrofitted to existing systems, without disruption to current database schemas, or application code [85]. It is considered a low intrusive approach because redo logs are a natural by product of most OTLP systems, kept for backup and recovery purposes [8]. The logs can be used for CDC, without needing to create an additional DBMS process, or modify the database schema, or modify existing application code.

However, building a tool to read the redo log is a potentially difficult task. This is because the format of the redo log is specific to each database management system. Furthermore, no standards exit for the format of the log, and vendors do not often document the format of their redo logs. However, some vendors (MySQL, Oracle, SQL Server) do provide an API that allows programmers to access the information inside the log. Despite such APIs, external log CDC can ultimately be difficult to implement from scratch. An alternative to building an external log CDC tool from scratch is to purchase an off-the-shelf

CDC tool.

There are myriad products available that offer the ability to do log CDC on a wide range of DBMSs. Most of the of the off-the-shelf products support the reading of redo logs for major database vendors such as: Ingres; IBM DB2; Microsoft SQL Server; Oracle; PostgreSQL; Sybase. Some systems also support legacy databases such as: IMS [8, 85]; VSAM [8, 53, 85]. Other systems though only support a limited range of DBMSs: Quest-Software [91] only supports Oracle databases; HVR Software [52] only supports Oracle, Microsoft SQL Server and Ingres; Flex CDC [41] only supports MySQL.

As well as being able to capture the changes from the logs, some tools offer additional features to support common tasks associated with CDC. HiTSoftware [51],IBM [53], QuestSoftware [91] and Wisdom Force [109] provide GUIs and graphical wizards that allow one to configure CDC. Attunity [8], Oracle [85] and IBM [53] offer the ability to wrap their tools in services, that are to be used in Service Oriented Architecture (SOA), for data integration purposes. HVR Software [52], Oracle [85], Attunity [8], IBM [53], Highleyman [50] support EBSs and message queues, which can be used to propagate and deliver changes to target systems. Oracle [85] and Attunity [8] offer filtering functions which allow targets to filter the change data on tables and rows, so that targets get only the data that they are interested in. HiTSoftware [51], Oracle [85], Wisdom Force [109] offer transformation tools, that allow one to specify mapping rules, which map data from heterogeneous schemas, into a target schema. In this thesis, we are specifically evaluating the core CDC mechanisms, in terms of real-time change identification and capture. The extra functionality is useful, but does not fall into the scope of the evaluation done in this thesis.

The creators of external log CDC tools claim that external log CDC is a real-time technology Attunity [8], Highleyman [50], HVR Software [52], Oracle [85], Wisdom Force [109]. This claim does not get backed up with empirical data. Although, Oracle [85] do claim that the latency with their tool is sub-second, and HVR Software [52] claim that their log CDC solution has a latency of less than a second. These figures are a vauge; it would be ideal to know the magnitude of latency in milliseconds. In Section 2.3.3, we argue that these technologies require mechanical latency, which limits their use for real-time CDC.

QuestSoftware [91] suggest that a drawback to some of the off-the-shelf tools is that they are complex and can be expensive to purchase. The high price for these tools could be justified by the two fold convenience they bring to an organisation. The first convenience is that they save an organisation's development time, by circumventing the arduous task of building a log scanner from scratch. The second convenience is that one can use these tools to retrospectively add CDC to their current systems. On the surface they are two useful benefits. However, the complexity of these tools means there is a learning curve to using them, which could be similar to the learning curve in understanding a log format and writing a scanner from scratch.

Being able to retrospectively fit log CDC to a system is useful, but the redo logs were

not originally intended for CDC. Redo logs do get archived offline, and log scanner could miss changes, if they archiving happens before the CDC scanning. Furthermore, these solutions have been built to be retrospectively fitted to relational database systems and legacy databases. There is a current trend in the field to move to NoSQL (Not Only SQL) databases, which have a non-relational data model. At this time these external log CDC tools do not support NoSQL databases.

2.3.2.4 Trigger Based Change Data Capture

Trigger CDC gets some attention in the literature [98, 65, 52, 99]. A database trigger is a piece of user written code that will automatically execute when given criteria are met. For CDC the firing criteria is a DELETE, INSERT or UPDATE DML statement that is applied to an operational table [98]. Moreover, the trigger will fire when changes are made to the data in a table that has triggers placed on it. Figure 2.8 depicts a pull CDC architecture, where triggers are used as the identification mechanism. When a change is made, the trigger will fire and replicate the DML statement on a staging table. The staging table's schema will be a replica of the operational table, but can also contain additional fields for CDC metadata [80]. A capture tool will be able to find the latest changes in the staging table.

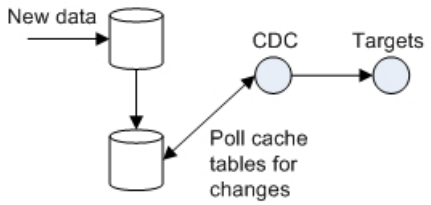


Figure 2.8: Architectual Overview of Trigger CDC

Like log CDC, trigger CDC is fairly low-intrusive towards current systems. One can create the triggers and staging tables, whilst keeping the current operational schema and data modification applications the same. Also like the redo logs, the staging tables are stored away from the operational tables, which means transactions are less likely to encounter contention due to read locks being in place from the capture tool's query. An advantage over external log CDC is that the capture tool does not have to be an off-the-shelf product. The staging tables will be SQL tables, which support SQL queries. Therefore, one can use the ODBC interface to capture data changes, without having to create or buy specialised tools.

Triggers identify changes, but are not capable of propagating the changes to targets outside of the database. A capture tool is necessary to capture the changes from the database and propagate them to targets. Although Norcot's [80] approach to trigger CDC, does allow one to create a publish/subscribe system that works in tandem with trigger CDC. Interested parties can subscribe to the staging tables, and a DBMS process will propagate the changes to targets. A limitation though is that the targets must be

oracle tables. If one requires the changes for analysis by some external application a capture tool will still be required, which is the limit of all trigger based CDC approaches.

We show in Section 2.3.3 that the capture tool requires mechanical latency, so that the capture queries not overwhelm the DBMS with requests. The mechanical latency will prevent real-time CDC. The only empirical data on the latency in trigger CDC is found in HVR Software [52], where they claim that the latency is about one minute. Again this data is vague, but it suggests that low latency is not possible with triggers. However, it is still worth discovering what the minimum near real-time deadline is for trigger CDC.

A drawback of trigger CDC is that they place an overhead on transactions[8]. HVR Software [52] show that triggers place a 3% overhead on interactive OLTP systems, and a 15% overhead on batch systems. Transactional performance is critical in an OLTP system [81], such an overhead could potentially make trigger CDC a non-viable solution.

Another drawback is that triggers need to be maintained and managed [8]. Moreover, as the database schemas and structure evolve over time, so must the triggers. The more triggers there are in a system then the more work one has to do in maintaining them.

2.3.2.5 Timestamp Change Data Capture

One can use timestamps to identify data changes. Timestamp CDC first requires one to modify the operational schema, so that each table where changes are applied to has an additional column to store a timestamp. The value in this field gets populated with the time at which the DML statement is applied to the table. Figure 2.9 shows the architecture of a CDC system where one uses timestamps to identify changes. In this approach, the capture tool will directly query the operational data tables for changes. The query will seek tuples where the value in the timestamp column is set to a later time than the time at which it the previous poll query had finished.

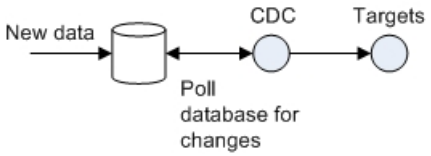


Figure 2.9: Architectual Overview of Timestamp CDC

Timestamp CDC is the cheapest approach to acquiring a CDC solution. One does not have to purchase expensive off-the-shelf products to use this type of CDC. Furthermore, there is a low learning curve to implementing timestamp CDC; one does not have to understand how triggers work, or have to learn how an off-the-shelf product works.

Timestamp identification is also quick. Changes are identified as soon as they are applied, because the timestamp column is populated as the tuple is created or updated. This is in comparison to log CDC, where a change has to be written to a file first, and trigger CDC, where the DBMS has to process a trigger before a change is identified.

Despite being a cheap and easy technology to learn, timestamp CDC has several draw-

backs. First, to expedite the capture query, indexes are placed on the timestamp column. This index has to be rebuilt when tuples change [63]. The index will constantly be changing, which may place an overhead on each transaction. Associated with this is the performance hit that is incurred by having to store the timestamp for every tuple that is inserted or updated [81].

Timestamp CDC is moderately intrusive, more so than triggers or log CDC. As well as adding a timestamp column to every table in the schema, one must also re-program the data modification applications to get them to populate the timestamp field, otherwise changes will be missed. This is less of an issue when inserting a new tuple, as one can instruct the DBMS to auto-populate the column by default with the current system time. However, on most DBMSs one must explicitly specify in the update statement, which columns are to be updated, this includes the timestamp column. Re-programming application code could be difficult, particularly if there are a lot of applications to change.

The biggest flaw of timestamp CDC, is the fact that it will not be able to capture changes that involve deleting a record [58]. When a record is deleted from the database, the data in the timestamp column is also deleted. In correlation to that, a change can be missed if the timestamp is overwritten before the capture tool can extract it. For example, a newly inserted tuple will have a given timestamp, but if an update statement is applied to this tuple then its timestamp value will change. If the update is applied to the tuple before the capture tool runs its query, then the first change will be lost.

Like with the other pull CDC technologies, it is unlikely that timestamp CDC can be used for real-time CDC. Again a capture tool that is external to the database is required to extract changes and propagate them to the target. In Section 2.3.3 we show that this capture tool will require mechanical latency. Finally, no empirical data has been published on the capture latency that exists in timestamp CDC.

2.3.3 Pull Change Data Capture for Real-Time

Change data capture is seen as a technology that enables real-time CDC. Whilst it is true that CDC can supply targets with fresh data more frequently than batch ETL systems; we present in this section an argument for why pull CDC architectures are unable to produce real-time CDC, according to Definition 2. Furthermore, mechanical latency is required in the capture tool, so the database can maintain a high transactional throughput. Moreover, the identification mechanisms on which pull CDC is built cannot propagate the changes to targets. Therefore a capture tool is required in order to extract and deliver the identified changes. We show theoretically that mechanical latency needs to be introduced into the capture tool, so that the DBMS can maintain the high transaction performance, which is critical in an OLTP environment [81]. A real-time CDC approach, without mechanical latency, ultimately requires a change identification mechanism that does not rely on relational DBMS features.

This section also highlights the exiguous amount of empirical data there is on the effect of using pull CDC and high frequent polling to do real-time CDC. Empirical data

is required to solidify the theory that is discussed in this section.

In pull CDC architectures each of the change identification techniques rely on a mechanism that is provided by the DMBS. Identified changes are stored in a change resource such as a staging area, or a redo log, depending on the mechanism used. In order for these changes to be used by target systems, such as fraud analysis engines, or data warehouse transformation and loading tools, a capture tool must extract the changes from these DBMS resources.

The capture tool is external to the DBMS, so it will have to poll the change resource to discover whether new changes have been made. Poll requests seldom run successively. Rather, there are intervals between each poll operation, so that the resource being polled can be used by other systems. The polling interval is a mechanical latency.

To better understand the need for mechanical latency, let us first consider what a poll operation entails. A poll operation is essentially a read request on a change resource. In the case of trigger CDC, timestamp CDC and DBMS log CDC, this read request will be a SQL query. In the case of external log CDC, the read request will be a file read request on the database redo log. All of the read requests will be seeking changes that have not been previously read by the capture tool. If new changes have occurred, then the response to the read request will be a set of changes. The poll operation can then pass this change set to a change processor, which will send the changes to the appropriate target systems.

The read requests on the change resources are simultaneously competing with write requests that want to change the resources. OLTP databases have heavy workloads that are often skewed more to writing data than reading it [45]. Transactions mainly involve writing data, so the more frequently changes are written to the operational tables, the more frequently there are write requests to a change resource. Furthermore, good OLTP systems have ACID transactions where the isolation property is set to a level to prevent issues like: lost updates; dirty reads; non-repeatable reads; phantom reads. Locking mechanisms are used to prevent these issues [93]. Locking up a resource to read change data could block write requests to the resource. If the poll operation were to run successively, without pause, it could continuously place a read lock on a change resource, thus potentially blocking write requests. Write requests will then build up in the DBMS's write buffer. If the buffers become full, then there will be a knock on effect towards transactional throughput. To prevent this, mechanical latency, in the form of an interval is placed between consecutive poll operations. The mechanical latency will mean there is less lock contention in the database and there be enough time for the writers to write data, before the buffer overflows.

The amount of mechanical latency required depends on real-time requirements versus the transactional throughput. Moreover, read locks versus write locks. Highleyman [50] report a fraud analysis system, where the DBMS has to handle up to 5000 transactions per second. If each transaction writes to say just three tables, then that is 15000 data writes per second to the operational tables, and also 15000 writes to the change resource per second. It would also mean there would be 15 data changes every millisecond.

Table 2.1 shows the cost of polling at various frequencies, in terms of the number of CDC data reads per second and the number of changes that occur in a given interval. When the interval between poll operations is 1 second, there will be 3 CDC data reads per second, one for each staging table[1], or in the case of external log CDC, there will be 1 CDC data read on the redo log file per second. A maximum of 3 CDC data reads per second is unlikely to impact the transactional throughput. However, during this one second, 15000 data changes will have been applied to the database. Processing 15000 changes will require a certain amount of processing latency. In a real-time system such as a fraud analysis system, this processing latency plus the mechanical latency may mean that the value of the change data is lost by the time a useful decision can be made on the data, as fraudulent activities could already have occurred.

Poll Frequency (ms)	**Reads Per second on staging tables**	**Reads Per second on redo log**	**Changes To Extract**
1	3000	1000	15
10	300	100	150
50	60	20	750
100	30	10	1500
500	6	2	7500
1000	3	1	15000

Table 2.1: The Cost of Polling in Terms of CDC Reads and Number of Changes that Occur in a Given Interval

One can see from Table 2.1 that the more frequently the poll operation runs the less changes there are to extract, which means the extracted change sets can be processed faster. Moreover, at the higher poll frequencies, there will be less mechanical latency as well as less processing latency, which means there is more time to make a valuable business decision, before the opportunity is missed. The consequence of a higher polling frequency is a higher number of CDC data reads, and thus more data read locks. For example, when polling once every millisecond, there will be 3000 CDC data reads per second if the change resources are staging tables, or 1000 CDC reads per second if the change resource is a redo log. This is a substantial number of data reads which could affect the transactional throughput. The additional CDC data reads will cause contention in the database. This contention, as previously mentioned, could block data writes, which will consequently reduce the transactional throughput. Therefore, the trade off with a pull CDC mechanism is a high transactional throughput versus low latency CDC. A high transactional throughput is more desirable in an OLTP system [81]. This is why there is a need for mechanical latency, which ultimately prevents real-time CDC.

One could argue that the mechanical latency could be lowered by upgrading the system hardware. Increasing the speed and memory capacity would allow one to increase the size

[1] each operational table has a staging table

of the DBMS buffers, which means they would take longer to fill up. A faster hard disk would allow for faster I/O, which means there will be less contention, because the length of time that write and read operations hold locks will reduce. A faster CPU will allow things like triggers, index look ups and rebuilds, and DBMS log readers to run quicker, which will reduce the impact these mechanisms have on the transaction rate. However, hardware does have its upper limits. Whilst Moore's law applies to CPU power, it applies less to memory and hard disk speeds [15]. Burleson [15] foresees that database administrators are going to continue to see hard disk and memory bottlenecks in the coming years.

When upgrading the system's hardware, to enable faster CDC, one is still likely to encounter the same problems eventually, if the business grows. Increasing hardware is reacting to a problem, rather than removing a problem. A better CDC solution would be to re-engineer the CDC process so that changes are stored in a resource that is not connected to the operational DBMS. The benefit of a solution that does not use the DBMS to identify changes will be the removal of mechanical latency, and thus the gaining of a real-time system.

Despite our argument for why pull CDC cannot be used for real-time CDC, there is little empirical data to determine the cost of high frequent polling. It would be worth knowing how much latency is in a pull CDC system, and what the empirical cost is to the transaction rate when polling at a high frequency. Few works in the literature have investigated the trade-off between transaction throughput and polling. Various works have shown the impact individual CDC mechanisms have on transaction rate.

HVR Software [52] show the impact trigger CDC has on OLTP systems and batch loading systems. We have previously reviewed this work in Section 2.3.2.4. This work highlights the overhead of using triggers to identify changes, rather than the I/O overhead, which is induced by CDC polling when capturing changes.

Pareek [88] attempts to obtain a quantitative evaluation of the overheads associated with CDC. They do this by measuring key performance metrics, such as: CDC effect on transactional throughput; CDC effect on redo overhead; CDC effect on system CPU. Pareek takes these measurements for Microsoft's SQL Server 2008, internal log CDC. Pareek used the TPC-C benchmark to evaluate this type of CDC. For a baseline, they additionally do an experiment where there is no CDC acting on the database. The results from Pareek's work shows that internal log CDC, has a negative impact on transaction rate; it increases the I/O overhead on the redo log; it requires more CPU power. Pareek's work confirms our theory in that he shows an increased I/O overhead. However, Pareek only does this for DBMS log CDC. Furthermore, Pareek's work evaluated CDC in only one scenario. A better evaluation would have run experiment scenarios with a wide range of transaction workloads. At different workloads one could have full appreciation of the I/O overhead. For example at a lower transactional workload, it is possible that higher polling frequencies can be tolerated, because there will be less competition for locks. Finally Pareek's work determined the cost of using DBMS log CDC to do change identification, but not the cost to extract these changes in real-time.

Raitto [92] provides empirical data on the performance of Oracle trigger CDC and Oracle DBMS log CDC. Raitto's work measures the impact these mechanism have on the transaction rate, as well as the impact they have on CDC latency. Raitto's CDC latency results show that half of the change data is captured in less than a second. However, Raitto uses data capture to refer to what this thesis defines as data identification. Moreover, the capture latency measured by Raitto, is actually the latency to identify the changes. A capture tool is still required to capture these changes, if the changes are going to be used by some external process. The measurements Raitto makes on transactional performance are similar to HVR Software's in that they measure the mechanisms impact on transactional performance when identifying a change, rather than the impact of polling when capturing the change.

Ultimately, there is a limited amount of empirical data in the literature on the I/O overhead associated with polling. The field would benefit from a comprehensive study into pull CDC latencies and the impact pull CDC has on the transaction throughput. Conn [25] provided a review of feasible implementation methods and architectures for real time data analysis, one of his conclusions was that his research is limited by the availability of performance metrics on an implemented system. Attaining a wider range of performance metrics on: CDC latency; transactional overheads; CPU usage, will allow us to determine the impact of using pull CDC in a real-time environment, and to gain an understanding about when they should be used.

2.3.4 Push Change Data Capture

Push CDC identifies and catches changes without the aid of a DBMS resource. Moreover, a push CDC mechanism does not rely on the database functions to do change identification. Instead, a push CDC mechanism will identify the data changes before they are applied to the database. Once the changes have been successfully applied to the database, the push mechanism can then push the identified changes towards some target. Push CDC is made up of two constituent processes: 1) change identification; 2) change delivery.

2.3.4.1 Change Identification

Change identification is the process that works out whether or not a change has been made. The concept of push CDC is to identify the change before it is entered into the database. There are currently two ways of doing this: 1) monitoring network traffic that is destined for the database server; 2) placing change identification code inside the application that is making the change. Both of these are discussed in more detail in Sections 2.3.4.3 and 2.3.4.4 respectively. The two mechanisms are fairly similar in that they both keep track of changes that are being sent to the database. It is important that the identification process does not assume that the database is passive. In other words, the identification process must receive acknowledgement that the change has been applied to the database, before it can flag the change for delivery. Without such a process in place,

the changes that were not successfully applied to the database will be propagated to the targets, thus invalidating any subsequent analysis that involves these changes.

2.3.4.2 Change Delivery

A push CDC mechanism is more likely to include a delivery process that will deliver changes to target systems. This is because the network sniffer or the data modification application are unlikely to be the target applications that process change data. Rather analysis engines or data warehouse transformation and loading tools will be the targets of the change data. The locations of the targets can be hard wired into the applications that are identifying the changes, or one of the more sophisticated change delivery mechanisms could be used, such as the publish and subscribe pattern or an ESB, mentioned in Section 2.1.2.1.

2.3.4.3 Network Based Change Data Capture

One can identify data changes by monitoring traffic on a network. There are tools, called network sniffers, which are able to filter and record network traffic that matches a certain pattern. Kimball [63] recognises that by using a network sniffer, one can monitor traffic heading towards a database server. This traffic can then be filtered to find requests that are being sent to modify the database. One can piece these packets together to determine the DML statements that are being applied to a database. This technique is similar to log CDC, in the sense that one is determining the changes, by reverse engineering the DML statements that were applied to the database.

Network based CDC, works by sniffing data packets that are travelling on the network, and collecting all traffic that is destined for the database. The traffic will be allowed to continue towards the database, but the network sniffer will also keep a copy of the traffic, so that it can be analysed for data changes. In order to ensure that the database is not passive, the network sniffer will also have to monitor and record database's response to each request. The databases response can be used to determine if the change was successfully applied to the database. Using this approach, one does not have to poll a database to determine the changes that have been made.

Network based CDC works; providing that the client and database communicate via plain text messages. Network based CDC will fail if the client is required to use an encrypted connection to communicate with the database. Only the database and clients will be able to decrypt these network packages. The network sniffer will not have the appropriate keys to decrypt the packets, and will therefore, not be able to identify which packets are DML operations.

2.3.4.4 Application Change Data Capture

An approach for doing push CDC is to place the CDC code inside the applications that are making the data changes. Several works recognise that an application can be used

to CDC [36, 8]. The data modification application may seem like a logical place to put CDC as the application knows what changes it is making to the database, which makes it an authoritative source of change data. Figure 2.10 depicts how application based CDC works. Each application (A1-A5) that interacts with the database will contain a routine for capturing the data changes that are being sent to the database. Essentially, this routine need be nothing more than a few variables to hold the change data that is being sent to the database. The application will communicate with the database through some form of ODBC. Once the change has been made on the database, the ODBC will inform the application as to whether or not a change has been successfully committed. As soon as the application has received an indication that the changes have been committed, then the application can immediately pass these same changes onto the targets.

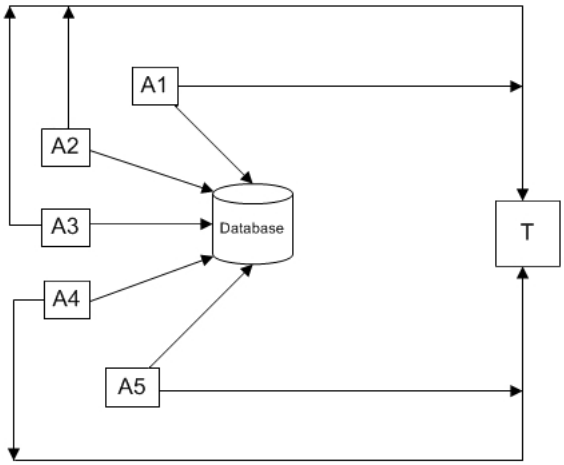


Figure 2.10: Architectual Overview of Application based CDC

Application CDC is faster than pull CDC techniques, because there is only processing latency between a change being successfully committed to the database, and a change being propagated to the targets. Application based CDC does not require mechanical latency because it obtains the change data from within its own memory context, rather than a change resource that is connected to a DBMS. Furthermore, memory I/O is faster than hard disk I/O. Application based CDC eliminates expensive hard disk I/O from the CDC process, and thus overall latency is reduced.

Despite the fact that application CDC can reduce capture latency, it is seldom implemented by practitioners. There are no case studies in the literature that detail the use of application CDC. Although it is recognised that push CDC is the key to reducing capture latency [88]. There are a number of valid reasons for why application CDC is rarely implemented.

One reason is that application CDC is highly disruptive towards current systems. It is difficult to retrospectively fit application CDC into existing systems. Attunity [8] argue that there are likely to be hundreds of applications that make changes to the data. The cost of changing all of these applications to include CDC routines is prohibitive. The pull

CDC systems are a lot less intrusive towards current systems; often at most requiring schema modification, to include timestamp columns on operational tables.

In relation to the intrusion issue, is the maintenance task one will face, once each application has CDC routines. Should the database schema evolve, or new targets require the change data, then each application that interacts with the database will need to be modified. In essence the applications are too highly coupled to the CDC code, thus creating an arduous maintenance task. Additionally, upon modifying an application, one will have to have a strategy in place to distribute the recompiled version of the application to all users of the system. Otherwise, one could face a situation where some users are using an out-of-date application, which does not have the correct CDC logic, which means crucial changes could be missed.

2.3.5 Improving Push CDC

Push CDC is the way forward if one wants real-time CDC. However, current solutions to push CDC are heavily flawed. Aside from application CDC, and network CDC, no other solutions for doing push CDC have been proposed.

To improve push CDC, and make it a viable solution, one will have to overcome the aforementioned intrusion and maintenance problems associated with the technology. A viable solution to push CDC will:

- minimise the level of intrusion on current systems
- Be easy to maintain.
- Keep the DBMS resource consumption to a minimum

We propose such a solution in the next chapter. Our solution will utilise a Service Oriented Architecture (SOA) to limit intrusiveness and minimise future CDC maintenance tasks. Moreover, we shall propose placing CDC logic into discrete services, which will encapsulate data modification code. These CDC services will be responsible for all data access and modification operations, as well as being able to do change capture, and change delivery.

2.3.6 SOA Usage in Data Warehouse Systems

We are not the first to consider using SOAs to advance data warehouse systems. This section, reviews other work that use SOAs to do a variety of tasks across the data warehouse spectrum, and describe how our contribution fits in with that work. Furthermore, services in a SOA seldom act alone, and instead, typically form a part of a greater task [35]. This section also shows how push CDC services could complement existing services.

Wang and Liu [107] proposed and implemented a SOA based real-time data warehouse. They use services in all parts of the data warehousing architecture. This includes a service for doing change data capture. Their data extraction service, called “data gather service”,

gathers data from the operational sources [111]. Like others, they recognise that change data capture is vital to producing real-time data warehouse architectures, and propose that their data gathering service can be used to facilitate a real-time data warehouse.

Jun et al. [56] provide details on the implementation of these data gathering services. The data gathering services are based on pull CDC. They do CDC by creating triggers on the source tables, and use the services to pull the data changes via a polling query. Their web services then distribute the change data to the rest of the architecture. This is different to the CDC services that we shall propose, as we shall propose to do the data capture within the services, as well as change distribution. The approach taken by Jun et al. [56] is fundamentally a pull CDC mechanism with a service interface. Therefore, mechanical latency will most likely be in this system, thus preventing real-time CDC.

Despite Jun et al.'s data gather service not facilitating true push CDC, their work still offers an effective solution to the data warehouse loading step. Not all change data that is captured in real-time is needed by the warehouse in real-time. To handle this, they propose a novel cache based queue system that can ease the burden of data loading. They provide four caches with warehouse loading cycles of 0 minutes, 10 minutes, 30 minutes and 60 minutes [107]. With this system the data that is vital for real-time operations can be loaded into the warehouse immediately, and data that is not vital for real-time operations can wait. This sort of priority based loading system would complement our CDC system, because we too would be generating copious amounts of real-time data, which might not all be necessary for real-time operations.

Finally, the work presented in the papers by Wang and Liu, Zhu et al. and, Jun et al. do not provide any empirical data on the transactional performance or capture latency in their CDC solution.

Pintar et al. [89] proposed a metadata-driven SOA based application for the facilitation of real-time data warehousing. In Pintar et al.'s solution, each part of the ETL process is coupled with a specific web service. The aim of the architecture is to have metadata-driven web services, which will allow for an automatic, reusable and retunable architecture for loading a real-time data warehouse, or to supply other applications with real-time data updates. The service most related to our work is the CDC extraction service. The discussion of the extraction service and the event detection to start the extraction is outside of the scope of their paper. However, they do mention that their extraction service pulls data from the JDBC's Database MetaData object. We assume that they are talking about pull CDC. Their web service CDC solution is similar to the one proposed by Jun et al. [56], and will again most likely require mechanical latency. It is possible though, that our solution to CDC could benefit Pintar et al.'s infrastructure by providing faster CDC, which will shorten the overall latency in their system.

Another approach to SOA based data warehouseing is proposed by Nguyen et al. [78]. Nguyen et al.'s work is different to the others in that it is using SOA to facilitate a real-time fraud detection system, rather than set up a data warehousing solution. The goal of their architecture is to provide real-time data for a fraud analysis system. They

have services for completing a sense and respond loop. Their architecture has 5 phases: 1) sense; 2) interpret; 3) analyse 4) decide; 5) respond. The architecture incorporates services that are responsible for performing duties in each of these stages. The stage most interesting to us is the sense stage. They do not provide a comprehensive insight into the implementation of this service, except to say it feeds their SARESA (Sense and Respond Service Architecture) architecture. They mention capturing events from source systems, which suggests they use pull CDC.

Schlesinger et al. [97] promote the use of web services to reduce the cost of ETL, in particular the cost of the data integration effort. Schlesinger et al. recognise the large effort required to integrate data from heterogeneous data sources into ETL systems. To reduce this effort they draw on SOA software engineering principles, by exposing data sources with a set of web services. Schlesinger et al.'s services are reusable across the whole architecture which reduces development costs as well as eases data integration. We too plan to utilise services in a similar way to reduce the development and maintenance costs associated with push CDC. Schlesinger et al.'s services are different to the ones we propose, because they only use services to support ETL tasks, by providing a clean interface for accessing a database. Whereas we will do the data extraction task within a service.

HVR Software's [52], Attunity's [8], and IBM's [53] log CDC tools can be exposed by SOA services. SOA based CDC solutions are easier to integrate with ESBs [8, 53] and publish and subscribe mechanisms, where changes can be published as change feeds [52], which means the locations of the target applications do not have to hard coded into the CDC logic. Furthermore, SOA interfaces on CDC tools help to ease data integration by offering a common interface for data access logic [52]. A target may miss a set of data changes; a CDC interface can expose data access operations which will allow a target tool to later request any change data it has missed. Additionally services can promote a standardised data exchange protocol for change data [8, 53, 52]. Services often communicate by exchanging XML based messages. The change data can be marshalled into standardised XML documents; a standardised XML schema makes it easier for change targets to parse and integrate the change data. Our SOA based CDC solution can benefit from similar advantages. The difference between our work and these tools is that our CDC logic will be based on a push mechanism, where as these tools use pull CDC mechanisms.

2.4 Summary

This chapter defined a real-time data warehousing architecture as an architecture that only has processing latency when moving changes from a source system to targets that use those changes to make an informed business decision. The definition of real-time contrasts to the definition of near real-time; the aim of near real-time is to provide targets with fresh data when they need it, rather than continuously. In a near real-time system, fresh data extracts are separated by mechanical latency.

Researchers and practitioners are continuously aiming to improve real-time architectures, by reducing the latency that exists in them. We have identified that real-time change data capture has not received as much attention in the literature as other components of the real-time architecture.

This chapter provided an in-depth discussion of pull and push CDC architectures, and the CDC mechanisms that are associated with each of those architectures. Table 2.2 and Table 2.3 display a taxonomy of the CDC mechanisms associated with pull and push architectures respectively. Table 2.2 and Table 2.3 also summarise the advantages and disadvantages of each CDC mechanism, as well as giving a brief description for when a given mechanism is prefered over its alternatives.

CDC Mechanism	**Advantages**	**Disadvantages**	**Preferable**
Internal Log CDC	Captures additional metadata on change	Requires a capture tool to extract changes, which requires a polling mechanism	When one does not require real-time CDC, and has an Oracle or SQL server DBMS with resources to spare.
	An out of the box solution if one has Oracle or SQL Server	Spawns a process that uses DBMS resources	
	Oracle's system can group together a transaction's changes	Only available for Oracle and SQL Server users	
External Log CDC	A low intrusive approach to CDC	Redo logs can be difficult to parse	When one has existing systems that would be too difficult to modify. External log CDC will provide an efficient CDC solution, with little to no intrusion on existing systems.
	Easy to retrofit to existing solutions	Redo logs can be archived off-line	
	Plenty of off the shelf products are available to support log parsing	No support for NoSQL databases	
		Redo logs reside on hard disks, which can cause I/O bottleneck	
Trigger CDC	Most DBMSs support triggers	Requires a capture tool to extract changes, which requires a polling mechanism	When one does not have Oracle or SQL Server, and does not want to purchase a tool to do log CDC.

Continued on next page

Table 2.2 – continued from previous page

CDC Mechanism	Advantages	Disadvantages	Perferablem
	A relatively low intrusive approach to CDC	Triggers consume DBMS resources and place an overhead on transactions	
		Triggers require maintenance	
Timestamp CDC	A relatively simple approach to CDC	Requires a capture tool to extract changes, which requires a polling mechanism	When one requires only an austere CDC architecture
		One has to add timestamp columns to all of the tables of interest	
		Applications have to be modified, so that they update the timestamp when changing a record	
		Changes where records are deleted are not captured	

Table 2.2: A Taxonomy of Pull CDC Mechanisms and their Associated Strengths and Weaknesses

There are claims in some of the literature that the pull CDC technologies are capable of real-time CDC. However, most of these works neglect the fact that changes have to be extracted from the change resource before they can be used by the target systems. It is our hypothesis that mechanical latency will have to be introduced into pull CDC mechanisms so that the data extraction does not overwhelm the database. Ultimately this mechanical latency will prevent real-time CDC. There is exiguous data in the literature to determine whether high frequent polling will impact the performance of the OLTP database.

The alternative to pull CDC is push CDC. It is recognised in the literature, that push CDC can enable real-time CDC, because it does not need to poll DBMS resources. However, push CDC is limited, by the fact that it is highly invasive towards current system infrastructures, and is difficult to maintain. It is the aim of this thesis to produce a viable approach to push CDC.

CDC Mechanism	**Advantages**	**Disadvantages**	**Preferable**
Network	Eliminates polling	Fails if packets are encrypted	When one requires a push CDC solution, but does not want to modify existing systems
	Does not require DBMS resources to capture changes		
	A low intrusive approach to CDC		
Application Based	Offers a real-time solution	Highly intrusive towards existing systems	When one requires a real-time CDC system and can afford to modify existing systems.
	Does not require DBMS resources to capture changes	Arduous to maintain	

Table 2.3: A Taxonomy of Push CDC Mechanisms and their Associated Strengths and Weaknesses

The ultimate reason for producing a viable push CDC approach is to reduce latency. If one is to reduce latency, then one must first understand how much latency exists in current approaches. Therefore, as part of this thesis, we provide an empirical examination of pull CDC technologies. This will allow us to discover how much latency exists in these systems. Discovering the amount of latency in pull CDC will provide us with a baseline to compare push CDC to. We shall be able to gain insight into how much latency one can reduce with push CDC. This will help determine whether push CDC is the optimal approach for a real-time data warehousing.

Chapter 3

Web Service Change Data Capture

The previous chapter showed that theoretically, pull CDC architectures require mechanical latency, which prevents them from being used for real-time CDC. Our theory suggests that push CDC is required for real-time CDC, because it only has processing latency. The previous chapter discussed fatal flaws in push CDC architectures, which prevent them from being adopted in CDC solutions. The most feasible of the push architectures, is application based CDC. However, application based CDC requires applications to be redesigned, and redeployed [88]. If application based CDC was not so intrusive towards existing application code, and subsequently difficult to maintain, then application based CDC would be a feasible push CDC architecture.

The aim of this chapter is to provide a feasible push CDC architecture, by resolving the intrusion and maintenance issues. This chapter proposes that the paradigm shift towards Service Oriented Architectures (SOA) can help alleviate the issues of intrusiveness and maintenance. Moreover, we propose a push CDC architecture that is built around the sound software engineering principles of SOAs. We compare our proposal with previous work, in terms of architecture and software design. Furthermore, this chapter demonstrates how a SOA based CDC solution can be used for real-time CDC, and show how it can provide equivalent functionality to that, which is found in existing CDC solutions.

3.1 Service Oriented Architecture for CDC

A service oriented architecture is a software engineering methodology for developing decoupled, cohesive and reusable components that are accessible regardless of their location. In a SOA, logic is split up and packaged into discreet services. Each service plays a role in the overall architecture, by being able to carry out a sub section of work. Erl, an eminent researcher in the SOA field defined eight service orientation design principles: 1) standardized service contracts; 2) service loose coupling; 3) service abstraction; 4) service reusability; 5) service autonomy; 6) service statelessness; 7) service discoverability; 8) service composability [35]. Not all applications require all eight design principles, but a selection of them can be used to provide a sound system design methodology for push

CDC architectures.

A number of those principles can be used to implement a service based CDC solution that is less intrusive and easier to maintain. Moreover, push CDC can be improved through loose coupling. One can use this principle to decouple data access code from the application, which in turn will allow one to decouple the CDC code too. The abstraction principle preserves loose coupling by ensuring that there is a clean interface for the application to access the service. Furthermore the abstraction hides the underlying details of CDC from the application, thus allowing the CDC logic to be changed without causing significant disruption to existing code. The reusability principle has implications for future applications; meaning they can get data access and CDC by default at no extra development cost. The autonomy principle could be useful if exposing ACID transactions as a service. Each transaction could be interfaced by a service, and require no additional interaction on the client side, until the transaction is complete.

3.1.1 Using SOA to Implement Data Access Services (DAS)

We have briefly mentioned how SOAs can be used to guide a better design methodology for push CDC. In this section we discuss in depth how one can use these principles to build Data Access Services (DAS). Moreover, we discuss how these principles can and have been used to create services that provide applications with access to a database.

There are myriad of techniques for designing and implementing data access services [10, 31, 11, 64, 7, 6]. We discuss these works, in detail, in Section 3.2.1. For now though, we focus on a high-level overview of data access services to explain how they work, and the advantages they bring to the architecture in which they are implemented.

Figure 3.1 shows a high level overview of a DAS architecture. In this architecture, all application interactions with the database are mediated by a DAS. These interactions take on different forms depending upon how the services are implemented. However, in general, to interact with the database, the client application sends a data retrieval or data manipulation request to the service. The service will parse the request, and will use the parsed information to decide how to interact with the database on the application's behalf. Depending upon the client's request, the service will send either a SQL statement or a DML operation to the DBMS. The DBMS returns the results or status of the operation to the service. The service will then relay the relevant information to the client, whether that is a result set or the status of the DML operation.

A DAS will provide a clean interface, so that any, language independent, client will be able to interact with the service through this interface. A clean interface allows the client applications to regard the DAS as a black box, thus hiding of the implementation details of the database including the schemas, from the client applications.

An authoritative software engineer, Scott Ambler, has identified several benefits of having an effective database encapsulation layer [5], which include:

- reduce the level of coupling between the object schema and the data schema, thereby

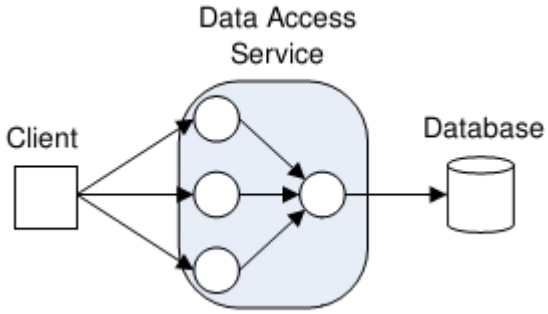


Figure 3.1: High Level Overview of a DAS

increasing the ability to evolve either one without affecting the other;

- all data-related code is in one place, making it easier to support any database schema changes or to support performance related changes;
- application programmers have a simplified job, as they will only have to deal with program source code, rather than program source code plus SQL code;
- a common place to implement data-oriented business rules;

Various other authors also recognise the aforementioned software engineering merits of encapsulation layers, and can extend Ambler's list:

- allow the development of light-weight clients that are optimal for devices with limited resources such as mobile phones and PDAs Deng [29];
- multiple data sources can be modified simultaneously Pais and Stancalie [87];
- data can be shared with remote users Pais and Stancalie [87];
- database drivers are hidden from the client, which allows one to change the drivers, without affecting the client code Koch et al. [64];
- a DAS allows the integration of heterogeneous data sources Hansen et al. [49], Abiteboul et al. [1];
- the DBA can create a DAS, and use it to control data access paths Naumann et al. [77];
- the encapsulated data source does not have to be a DBMS Anjomshoaa et al. [6];
- data access to heterogeneous data sources can be done through a common interface Anjomshoaa et al. [6]

A DAS can improve push CDC architectures by promoting better software engineering principles. The major benefit of DAS, for change data capture, is that with an encapsulation layer there is now a common location to implement data-oriented business rules and

CDC logic. Ultimately this removes the difficult maintenance task associated with application based, push CDC. Now, rather than have the CDC logic scattered across multiple applications, there is a single location for the CDC code.

3.1.2 DAS Based CDC

In Section 3.1.1 demostrated how a DAS improves the overall design of a system where applications interact with a database. In this section we demonstrate how a DAS can be used to do push CDC.

Figure 3.2 shows a high-level overview of how DAS-based CDC will work. The application provides the service with the change data, so that the service can apply it to the database. After the service has successfully applied the change data to the database, it can then forward the change data on, to a target system. It is important to note that the service forwards the change data only after confirmation is received from the database that the change has been successfully committed, which safe guards the service from forwarding change data that has subsequently been rolled back.

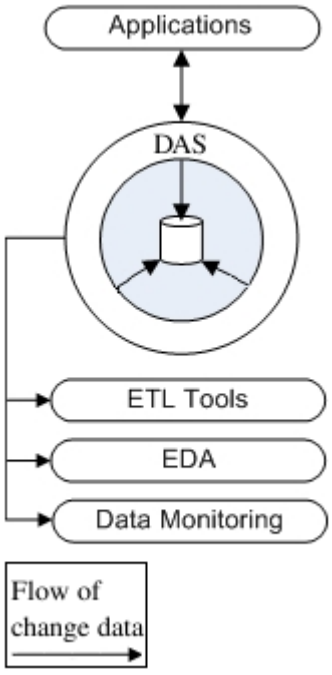


Figure 3.2: High Level Architecture of Our Proposed CDC system

The pseudo code in Figure 3.3 provides an insight into the internal working of this DAS-based CDC. In this example a method in the DAS called `Table1Modification()`, exposes some transactional logic. When the client calls the service, the service will pass the client's input data as parameters into the modification method. `Table1Modification()` will use the parameters to build a DML string (line 5). The DML string is then sent to the database for execution (line 6), also in this line the `execute()` method returns a boolean to indicate whether or not the DBMS has successfully applied the DML statement. An

`if` statement then determines the next course of action. If the condition is false, then the database is rolled back (line 9) and the method exits. In a real implementation it would likely return a message to the caller, indicating the reason for the failure. If the DML statement is successful, then the change, plus metadata about the change can be added to a change message (line 12). One can see that the change data is captured, without the need to poll the database. Here, the change data is captured from the DAS's memory context. Moreover, the change message is created from the parameters that were passed into `Table1Modification()`. Lines 15 - 23 repeat the change process for change2. If the code reaches line 25 without failure, an attempt is made to commit the transaction. Providing that the commit operation is successful; the change message can be sent to the target (line 31).

The target can then use this change data to do whatever it needs to do. In this example, the change message was a string of changes separated by a semi-colon. The target can tokenize the string on the semi-colon character, to obtain each individual change. Of course, the format of the change message can vary according to the system architect's discretion. For example the changes could be encoded in RDF, XML or JSON. To change the encoding of the changes, one would just have to reimplement the building of the change message.

Ultimately, DAS-CDC works in the same way as application based CDC, except the services are doing the data capturing rather than the applications. Therefore, DAS-CDC is push CDC, which according to our theory, is a requirement for real-time CDC. Moreover, the code example in Figure 3.3 shows that there is no mechanical latency involved in capturing the changes. The only latency is processing latency, so by Definition 2, DAS-CDC is a real-time technology.

```
1. boolean method Table1Modification(Change1, Change2){
2.       String changeMsg;
3.       dbConn = getDatabaseConnection();
4.
5.       dml = "INSERT INTO table1 (col1) VALUES (change1)";
6.       success = dbConn.execute(dml);
7.
8.       if(not success){
9.          dbConn.rollback();
10.         return false;
11.      }else{
12.         changeMsg = change1_metadata + change1 + ";";
13.      }
14.
15.      dml = "UPDATE table2 SET col2=change2 WHERE col1=1";
16.      success = dbConn.execute(dml);
17.
18.      if(not success){
19.         dbConn.rollback();
20.         return false;
21.      }else{
22.         changeMsg = changeMsg + change2_metadata + change2 +";";
23.      }
24.
25.      success = dbConn.commit();
26.
27.      if(not success){
28.         dbConn.rollback();
29.         return false;
30.      }
31.      sendChangesToTarget(changeMsg);
32. }
```

Figure 3.3: Pseudo Code for Capturing Two Data Changes

3.1.3 Dealing with Intrusiveness

We are advocating a push CDC solution that is based on good software design principles, in the aim to circumvent intrusiveness, and code maintenance issues. However, one could argue that DAS-CDC is still intrusive towards current systems, in the sense that existing applications still have to be modified, so that they interact with the DAS. Furthermore, the DAS may still have to be implemented. In this section, we put forward an argument for why DAS-CDC should be considered as less intrusive.

SOAs are becoming ever more popular in enterprise architectures. A study conducted by Evans Data Corporation [37], who provide in-depth survey of the global software development population, found that 68% of respondents say they are of exposing or invoking their database operations through service based mechanism. This suggests that DAS are

gaining popularity in industry. Additionally, Resource Oriented Architectures (ROA) are also gaining support in the community. A ROA is similar to a SOA, but services are called resources instead. ROAs can also be used to access data stores. The examples in Junye et al. [57], Pintar et al. [89], Daniel et al. [27], Cholia et al. [23] all use ROAs, where the organisation's resources[1] are accessed and modified through a set of web services. Work by Richardson and Ruby [96] dedicates a chapter designing ROAs. Finally, large companies such as Amazon [4], eBay [32], and Facebook [39] are providing web service APIs that allow access to their public data stores. Companies like this are often trend setters, who are followed by lesser known organisations.

Considering that many organisations are switching towards SOA based computing, compliments our DAS approach to CDC. With this paradigm shift the intrusiveness of DAS-CDC, is damped, by the fact that many organisations are switching to SOA based infrastructures. The paradigm shift towards SOA is pivotal in providing an apt solution to the issue of intrusion. Essentially SOA has made push CDC viable.

If this paradigm shift was not in place, then arguably the intrusiveness of DAS-CDC would be comparable to application based CDC. Applications would require changing so that they communicate with the DAS rather than the database. However, by taking advantage of good software design principles, DAS-CDC hides the implementation of CDC from the applications, which means once the applications know how to interact with the DAS; they are protected from future changes to the CDC code. Whereas if the CDC code was in each application, then every application would require maintenance when the CDC logic changes.

Organisations that are already using SOAs, especially to access data stores, are in a prime position to utilise DAS-CDC. To do this, they can modify their data access code to include CDC logic, and then provide a change data delivery mechanism.

3.2 Designing a DAS-CDC Solution

We presented our DAS CDC solution, as described above, to a group of researchers at IBM. The IBM researchers' biggest concern with our proposal was performance. The IBM researchers felt that DASs would place an overhead on a database's transaction rate, which would ultimately compromise transactional throughput. Essentially, one has to be diligent when designing DASs for an OLTP database, and ensure that the DASs do not hinder the performance of the OLTP database.

An organisation's OLTP environment will consist of a set of well-known, transactions and users [46]. These transactions will be ACID (Atomicity, Consistency, Isolation, Durability) compliant. Transactions in an OLTP environment must execute efficiently so that the DBMS can maintain a high transactional throughput, to meet the demand of its users. A DAS that mediates a data modification application and a database may place an additional communication overhead on the transaction, which may increase the time of the

[1] including databases

transaction. Therefore, a DAS for an OLTP environment must be designed to minimise the additional communication overhead. It is likely that others will share the IBM researchers' concern.

In this section we begin to address the concern by exploring possible designs for DAS, primarily in the context of performance. Our aim here is to find a DAS design and implementation that will place little to no overhead on the database's transactional throughput. In addition to performance, there are a number of secondary goals that a DAS must achieve, if it is to act a mediator for an OLTP database. The following list expresses all of the design goals that a DAS must meet, in order to be viable in an OLTP environment:

- Place little to no extra overhead on the time it takes to execute a transaction
- Use as little amount of bandwidth as possible in message communication, the theory being the less bandwidth used, then the quicker it will take to transfer the data, which in turn minimises the transaction time
- Use a communication format that the client and server will be able to parse with ease. Additionally, to keep the transaction time short, it will be advantageous to use a light-weight encoding format
- Minimise CPU time to marshal and unmarshal the communication messages
- The DAS will expose the database in an intuitive manner, that allows the developer to easily create applications that can use the DAS to execute a transaction
- The DAS should expose the database with security in mind: OLTP databases contain an organisation's operational data, some of which can be sensitive. If this data ends up in the hands of malevolent criminals, then organisation's reputation could be damaged, and business could be lost.

In the following sections, we shall explore the DAS literature to seek out a DAS design that may lead to a viable DAS implementation, in an OLTP environment.

3.2.1 DAS Design Methodology

There are two common, high level approaches to designing the interfaces for data access services: 1) Generic data access services; 2) Schema specific data access services.

The generic approach to exposing a database is popular in the literature and is implemented by Dogdu [31], Koch et al. [64], Deng [29], Katayama et al. [61]. Figure 3.4 depicts a standard generic interface for exposing a database. In a generic DAS interface, there will typically be operations, with signatures close to `query()` and `execute()` that allow a client to repsectively send either an SQL statement or a DML statement to the DAS. When a client invokes these operations, the DAS will attempt to execute the received, statement on the database. Once the DAS has executed the received received, it will return any necessary results to the client, or it will inform the client of any errors in the

event that an error occurs. Generic DAS are particularly useful in ad-hoc environments, where little is known about the users, or the types of queries they wish to execute.

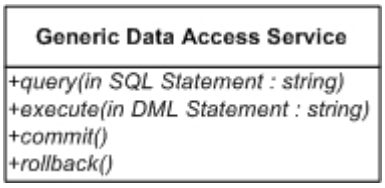


Figure 3.4: An Example of Exposing a Database with a Generic DAS Interface

Schema specific is a phrase we have coined, to describe data access services that expose operations or resources pertaining to a particular schema. Figure 3.5 depicts a schema specific approach to exposing a database. In a schema specific approach one exposes the database with operations that will specifically access or modify a particular schema. For example the interface in Figure 3.5 has operations such as `createNewOrder()`, and `productSearch()`, which are specific to an order entry schema. Or `debitAccount()` and `creditAccount()` to transfer money between bank accounts, in a bank schema. The operations that encapsulate DML logic require the client to pass change data in as parameters. The operations that encapsulate SQL statements, will take parameters to complete where clauses, or allow users to specify row limits, or ordering etc. Similar to the generic DAS, the schema specific DAS, will return any results or error messages after it has executed the invoked operation. Schema specific DAS offer the DAS developer stringent control over how the client can access and use the database. It is up to the developer to decide how much the clients can access or modify the data in the database.

There are few examples of schema specific DAS in the literature, the most notable example is by Borkar et al. [10]. Borkar et al. [10] shows how one can use E-R modeling to guide the implementation of the schema specific DAS. Each data service corresponds to an E-R entity, and provides read/write operations to access the entities attributes. For example, `getCustomer()` or `createCustomer()` operations that belong to a customer entity. Calling these methods will direct the DAS to execute a query on the specific entity. The nature of schema specific DAS makes them more suitable when one is building a DAS that are to be used by clients who have well-defined requirements regarding how they wish to access the database.

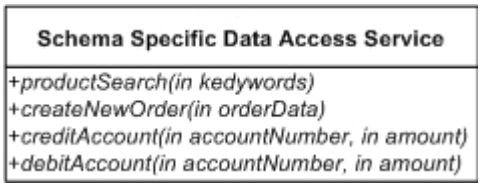


Figure 3.5: An Example of Exposing a Database with a Schema Specific DAS Interface

Table 3.1 presents a comparison of schema specific and generic DAS properties. By considering the properties in Table 3.1, we argue that a schema specific approach is more

suitable for an OLTP environment. The following section expands on this argument by specifically addressing these properties and discussing why they are desirable in an OLTP environment.

Property	**Schema Specific DAS**	**Generic DAS**
Can efficiently handle multi-action transactions	Yes	No
Level of coupling between database and clients	Low	Medium
Client interaction with DAS	Strict	Flexible

Table 3.1: Properties of Data Access Services

3.2.1.1 Client Interaction with DAS

OLTP environments have well-defined operations and users [46]. Any DAS that exposes an OLTP database need only expose the database schema enough so that these well defined operations can be executed. Moreover, a schema specific DAS is suitable to expose the operations that are pertinent to one's business operations. A generic DAS is more flexible, as they allow the client to send their own SQL string. This flexibility is superfluous when one has upfront knowledge about how the client will interact with the DAS. However, this flexibility is advantageous in scientific or analytical environments, where the client's queries are ad-hoc, and upfront knowledge of how the client will interact with the DAS is unknown.

3.2.1.2 Level of Coupling

A design that has a lower level of coupling is preferable, as it allows one to change the database without requiring the client applications to be modified. Both generic and schema specific DAS decouple the database from the clients that use it. However, generic DAS require the client to have knowledge about the database structure, i.e. table names and column names. Moreover, the client will have to know how to correctly format SQL and DML strings that will be semantically correct for the database that the generic DAS is exposing. Schema specific DAS fully encapsulate the database and hide underlying database structures and schemas. The client will only need to know which parameters to send in order to invoke a DAS operation, rather than needing to know about SQL or

DML. Considering that OLTP database operations are well defined [46], one can build the DAS so that the logic for accessing the database is hidden from the user, which promotes a lower level of coupling.

3.2.1.3 Enabling Efficient Execution of Transactions

The most important property of an OLTP database is that transactions execute efficiently. A DAS for an OLTP environment, should not have a negative effect on the transaction rate.

Theoretically, the generic modelling of DAS leads to a cumbersome way of developing services in an environment where efficiency is the primary objective. Moreover, transactions involving multiple actions would soon become inefficient. Considering the standard generic interface set out in Figure 3.4 the client would have to call a DAS operation (either `execute()` or `query()`) for every constituent action in the transaction. In other words a network round trip between the client and DAS would be required for every action. Figure 3.6 shows a sequence diagram for the execution of a two action transaction over a generic DAS. One can see from Figure 3.6 that each involves a round trip between the client and the server. There is also an additional round trip of network communication to commit the transaction.

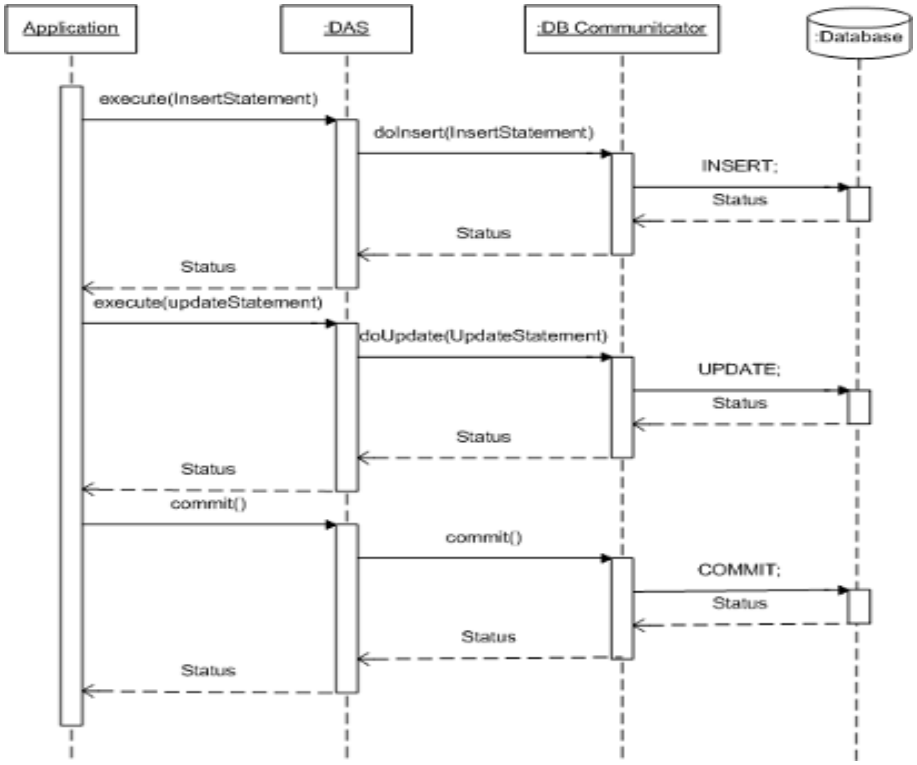


Figure 3.6: Sequence Diagram Showing Interaction with a Generic DAS

The communication overhead on each action will quickly accumulate and become a large overhead on the overall transaction. Deng [29] shows that executing a single action

through a generic DAS is three times slower than going direct to the database. Considering Deng's results, then each transaction action in the transaction would take three times longer, if one were accessing the database through a generic DAS. A large transaction with ten actions would be thirty times slower when accessing the database through a generic service. Such an overhead is undesirable in an OLTP system.

Schema specific modelling does not fully guarantee efficiency either. Should the DAS expose the transaction actions as individual operations, then like generic modelling, there will be a round trip of network communication for every transaction action. However, one can reduce the number of network round trips in the schema specific approach by exposing the entire transactional logic through an operation. When exposing the transactional logic through an operation, there will only be a single round trip of network communication between the client and the DAS. Figure 3.7 shows a sequence diagram for the execution of a two action transaction over via a schema specific DAS. One can see from Figure 3.7 that there is only one round trip of network communication between the client and the server. Once the client passes control to the DAS, the DAS will co-ordinate the execution of the transaction. By reducing the number of network round trips, the performance of the DAS theoretically improves.

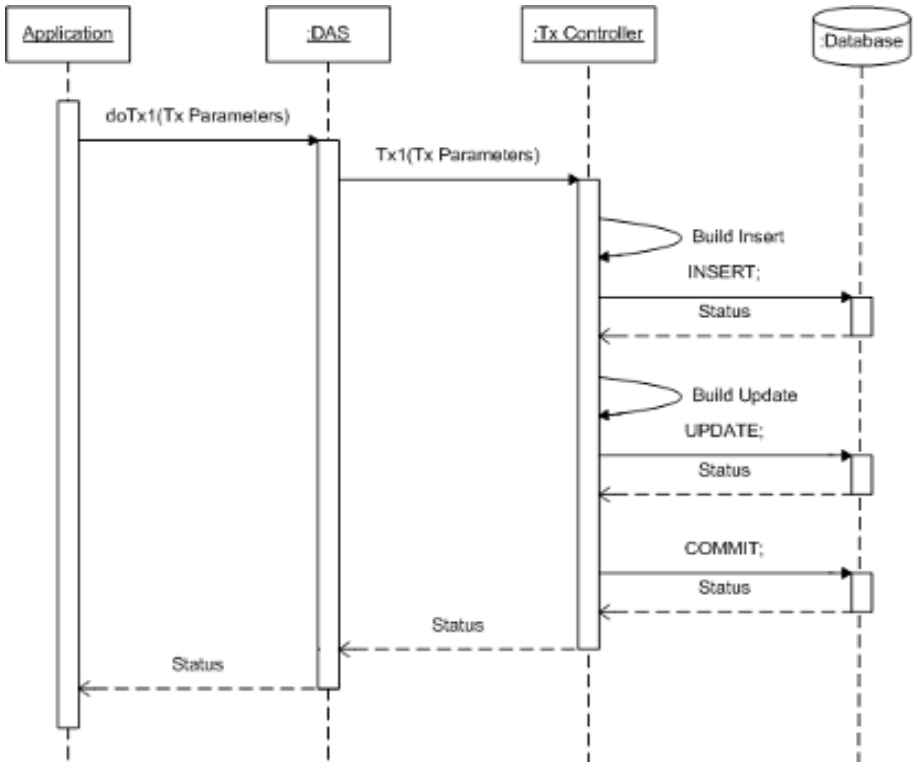


Figure 3.7: Sequence Diagram Showing Interaction with a Schema Specific DAS

In an OLTP environment it is possible to expose the entire transactional logic, because of the well defined nature of OLTP databases. The upfront knowledge about the transactions means, one will know before the DAS is built, what logic each transaction will contain, and also what the inputs and outputs of a transaction will be. Ultimately, in

terms of transaction performance, a schema specific approach that exposes the complete transactional logic will theoretically offer the most efficient way of creating DAS that are suitable for an OLTP database.

3.2.1.4 DAS Security

Both the schema specific and the generic approaches can be made secure to prevent unauthorised access, and deny users from carrying out actions that are beyond their permission level. A good example of a permission policy in a generic DAS is implemented by Koch et al. [64]. Here they specify different access levels, and only the clients with the right access levels can have their SQL or DML statement executed on the database. Similar security can also be implemented in a schema specific approach, so that only certain clients will be able to invoke the exposed operations.

If security is paramount, then further precautions can be taken. One may create and verify their own SSL certificates, and use HTTPS to transfer the data to the DAS. This will ensure that the DAS are protected against packet sniffers. Message encryption and decryption overheads may slow a transaction down when using HTTPS. However, the overhead would likely be similar if one were to use and encrypted channel of communication between the ODBC and the database.

Overall, both generic the schema specific approaches have their vulnerabilities. However, if a developer takes enough care, they can both be made secure enough to run in an OLTP environment.

3.2.1.5 DAS Message Format

The majority of the works in the literature are SOAP based, thus there is a heavy dependency on XML as the message format. XML is a verbose format to describe data exchanges. A light-weight alternative known as Java Script Object Notation (JSON) is being advocated as the message format in data exchange environments [23]. JSON is optimized for data, whilst XML is better as a document exchange format [55].

With performance efficiency being paramount to DAS in an OLTP environment, we feel JSON is the message format to use. This is because it will take up less bandwidth on the network, and thus there will be less of an overhead on the transactions. Geer [44] shows that a more proprietary data exchange format such as binary XML could have been used to gain further time saving. However, this begins to limit the number of programming languages that can decode the format. JSON parsers and encoders are widely available for all the major languages.

3.2.2 DAS Implementation Techonolgy

There are two technologies that are commonly used to implement services; they are SOAP-based services, or RESTful services.

The majority of the DAS implementations in the literature are SOAP-based. This is likely because the majority of the DAS are deployed in distributed scientific environments. The SOAP framework is heavily standardised. The standards make services open and accessible to anyone who understands the standards, which is beneficial in a large, distributed community. Furthermore, SOAP based services are said to be more suitable for analysis, where operations run for longer periods and often require complex parameters [61], which is another likely reason for their adoption in scientific communities.

SOAP also has standards such as WS-AtomicTransaction [83] and WS-BusinessActivity [84]. These standards are used to obligate two-phase commits and relax transaction isolation properties. Relaxing isolation properties is particularly useful in a distributed environment, where transactions are cross-organisational, and occasionally run for lengthy periods of time [3]. However, OLTP environments tend to be specific to one organisation, rather than distributed. Thus OLTP DAS are less likely to require the SOAP transaction standards. Furthermore, because OLTP DAS are specific to each organisation, there will be less need for the SOAP standards in general. Moreover, the DAS encapsulation layer will only be accessed by the organisation's business code [5], so the organisation can define their own in house standards that might be more efficient.

RESTful implementations are recommend when, when one has control over how the services are built and accessed [96], which, as previously mentioned, is likely to be the case in an OLTP environment. RESTful services do not have standards. However, Roy Fielding [40] set out a number of RESTful principles one should adhere to when creating RESTful services. By sticking to these principles, one can create clean uniform interfaces.

RESTful DAS implementations are also found in the literature. Daniel et al. [27] advocate RESTful services. Daniel et al. use RESTful services, because the four HTTP verbs (`POST`, `GET`, `UPDATE` and `DELETE`) provide a clean interface for the CRUD pattern to aid the Creation, Reading, Updating and Deletion of data. The Spitfire project introduced by Wolfgang and McCance [110] implement DAS around HTTP, which is the foundation of a RESTful service. However, Wolfgang and McCance's approach is not strictly RESTful, according to Fielding's principles, as both resource scoping and method information is placed in the URL. Wolfgang and McCance's approach is a REST-RPC hybrid, a phrase coined by Richardson and Ruby [96]. Katayama et al. [61] devise a generic RESTful DAS. Katayama et al.'s TogoWS provides a uniform query interface. Each URI can be mapped into an SQL string. Katayama et al.'s work expose everything as a resource, and each resource is an addressable via a URI, which is good RESTful design. Such a design could be copied for OLTP DAS by exposing transactions as a resource.

Richardson and Ruby [96] describe a methodology for creating RESTful transaction services. However, Richardson and Ruby's RESTful DAS expose each transaction action as a resource. Previously we have described the performance savings one can gain when the DAS exposes the whole transaction rather than each action of the transaction. When using Richardson and Ruby's approach there will likely be a performance issue. To consider this further, Table 3.2 shows an example of of Richardson and Ruby's DAS approach, for

a transaction consisting of two actions. Firstly, a client will call the `POST` operation to create a new transaction resource. The service then returns a URI pointing to the location of this newly created resource. The client then calls the `PUT` operation to get the service to invoke the first transaction action. The `PUT` operation essentially allows one to update a resource. In this case the client is updating the transaction resource by telling the service to execute one of the transaction's actions. The second action is then called via the a second call to the `PUT` operation for that particular transaction resource. A final call to `PUT` is made telling the DAS to update the resource by committing the transaction. The problem with this approach is that it requires four round trips of network communication to do a relatively simple transaction.

Send request to create transaction	`POST` /transaction/Tx
Server returns transaction URI	201 Created Location: /transaction/Tx/123
Request:	`PUT` /transaction/Tx/123/Action1 data=action1 data
Request:	`PUT` /transaction/Tx/123/Action2 data=action2 data
Request:	`PUT` /transaction/Tx/123 commit=true

Table 3.2: RESTful Transaction Action as a Resource

Table 3.3 shows the HTTP calls needed when exposing the entire transaction as a resource. Firstly, the clients calls the `POST` operation to create the transaction. All of the data required for the constituent transaction actions is sent along with this initial request. Moreover, all of the data required to create the transaction is sent upfront. The server then uses this `POST` data to create the resource, and also conduct the entire transaction, without any further communications with the client. After the transaction is complete, the server returns the URI for that transaction resource. This approach requires only one web service call to execute a transaction. The DAS can assume that the client wants to have the transaction automatically executed. This behaviour is a possible violation to the RESTful principles, because the DAS assumes that the transaction is to be automatically executed and committed. The traditional way would be to have a subsequent `PUT` call to tell the transaction to commit. However, an additional round trip of network communication is saved, if one assumes the client wants to automatically commit the transaction. In an OLTP environment it may be natural to assume that one would want to execute the transaction immediately. Furthermore, by passing the transaction action data into the initial `POST` operation, one can implement a RESTful approach to transactions that is

more efficient.

Send request to create transaction	`POST` /transaction/Tx data=transaction data commit=true
Server returns transaction URI	201 Created Location: /transaction/Tx/123

Table 3.3: RESTful Transaction as a Resource

We propose that the efficient RESTful approach will be the ideal implementation technique. As beneficial as the SOAP framework and standards are, they are likely to place an additional overhead on web service calls [22]. With REST, all communication is done over HTTP, without wrapping the messages in additional protocol layers. Thus, there will be less data on the network, which will ultimately lead to a performance gain.

3.3 Summary

A SOA based CDC solution fulfils the objective of having a push CDC architecture that is easier to maintain. The CDC logic is now encapsulated in single, common location that is accessible to all data modification applications. If the CDC logic is required to change in the future, then one no longer has to modify every data modification application, and instead only the DAS needs to be modified.

On the other hand, the issue of intrusiveness is still open to debate. A SOA based push CDC solution can still be intrusive towards current systems. In the worst case scenario one will have to implement the DAS, and will have to modify all existing applications to use the DAS, which is less convenient than using one of the pull CDC mechanisms. However, building the DAS could be considered a one off cost, which can be justified if DAS CDC can deliver true real-time CDC. We further justify the DAS approach by considering case studies which suggest there is a paradigm shift towards organisations implementing DAS to handle data access and data modification. Those who have implemented DAS, or are in the process of doing so will be less burdened by the intrusiveness of DAS CDC.

Additionally, it could be argued that DAS CDC is less intrusive than application based CDC, especially with the maintenance burden reduced. Ideally though, a study that measures and compares the intrusiveness of the two approaches against say the intrusiveness of pull CDC architectures is desirable. We are not conducting that study ourselves because we felt that there is a greater need to validate DAS CDC with empirical performance data rather than concentrate on a study of intrusiveness. If the concept of DAS CDC can be proven, and can be shown to deliver real-time CDC over the pull CDC architectures, then an intrusiveness study can be conducted. Moreover, it is not worth studying intrusiveness if DAS CDC does not perform better than pull CDC.

Chapter 4

Preliminary Evaluation of DAS Performance

The previous chapter identified two methodologies for DAS implementation: 1) generic; 2) schema specific. The theory behind these methodologies was discussed, and it was hypothesised that a schema specific approach is appropriate for developing DAS for an OLTP environment. Furthermore, it was identified that RESTful DAS may be optimal for an OLTP database, in terms of performance. To test that theory, this chapter extends the argument between generic and schema specific DAS for OLTP databases, by exploring how the various approaches and technologies perform in an empirical experiment.

This chapter begins by providing an overview of the existing literature on DAS performance benchmarking. Our review shows that there is an exiguous amount data on DAS performance in an OLTP environment. This chapter aims to fill the gap in the literature, by providing an experiment that will give us empirical data to compare the performance of four DAS implementations: 1) Generic RESTful DAS; 2) Schema Specific RESTful DAS; 3) Generic SOAP DAS; 4) Schema specific SOAP DAS. Benchmarking these four implementations allows us to compare schema specific DAS to generic DAS, and SOAP based DAS to RESTful DAS, which ultimately allows us to find the optimal solution.

4.1 DAS Performance Literature

There have been numerous studies conducted on the performance of web services. We are primarily concerned with performance studies where DAS are used to interface a database. Before we review DAS performance, it is worth considering studies on web service deployment architectures, which are relevant to our work, because they can help us to decide on an efficient deployment platform.

The two deployment platforms that are often compared in the literature are the .NET framework, and the J2EE platform. Sun Microsystems Inc [101] compared the J2EE platform to the .NET framework, and found that the J2EE platform outperformed the .NET framework in all of four of their bespoke benchmark tests. Microsoft Corporation's

rebuttal [74] showed that the .NET framework matches, and slightly outperforms the J2EE platform, in the same four benchmarks. Microsoft Corporation [74] then go on to show the .NET framework "significantly outperforms" the J2EE platform, in a more "realistic" benchmark, which involves higher message payloads. Given that each organisation claims that their product has better performance, it might be wise not to give these results too much credence. Liu and Gorton [68] present a less biased piece of research, in which they compare the two frameworks and find that they have similar levels of performance. A more recent study by Velmurugan and Mohamed [106] compares the two platforms with four different configurations, and found that the .NET platform is more efficient on the basis that it uses less packets in data communication. Considering the literature as a whole, the two platforms are considerably similar to each other. Our decision on which deployment platform to use can be decided based on factors other than performance.

There have been a number of attempts to examine the performance of data access services; however, there are no empirical studies that provide an insight into how these services perform in an OLTP environment. Deng [29] sets up a simple experiment to measure the time it takes to insert a record into the database, via a DAS. For comparison, Deng's work additionally measures the time it takes to insert a record by going directly to the database, through the ODBC. Deng's work shows that inserting a record via a DAS is three times slower, than going through ODBC. A DAS that makes inserting a record three times slower would place too big an overhead on an OLTP transaction. Furthermore, Deng's results are not comprehensive enough to cover use cases that one may find in an OLTP environment. Dogdu [31] measures the response time for querying increasingly complex datasets. The results provide an insight into query response times, when querying a database via a generic DAS. Dogdu finds that the response time of a query is longer when one selects more columns, due to larger data sets being sent on the network. Dogdu's experiments, do not measure the performance of operations one would wish to perform on an OLTP database, such as the performance of multi-action transactions. Koch et al. [64] provide more substantial experiments, but again only explore the performance of executing single "insert and select" statements through a DAS. They show that their implementation provides a viable level of performance for a grid environment, which is used by a scientific community. However, their work only shows the performance of executing single SQL statements through the DAS. Again, a transaction often consists of multiple SQL statements and DML operations. Whilst Koch et al.'s implementation supports transactions, no transaction performance data is provided. Pais et al. [86] took a different approach to measuring performance, by measuring bandwidth usage, rather than measuring the time to execute a statement via a DAS. They show that going directly to the database used approximately half as much bandwidth than SOAP based DAS. Their work presents mechanisms such as caching and compression to reduce the overall bandwidth usage when using the DAS approach.

To be viable for use with an OLTP database, the DAS must not impact the database's transactional throughput. Because there is a lack of published empirical data on the

performance of DAS in an OLTP environment, we investigate the possible methodologies for implementing the DAS, and then propose a preliminary experiment to evaluate the performance of DAS in a transactional environment.

4.2 Preliminary Evaluation of DAS

We have previously identified two predominant methodologies for DAS implementation: 1) generic; 2) schema specific. The theory behind these methodologies was discussed, and it was hypothesised that a schema specific approach is appropriate for developing DAS for an OLTP environment. Furthermore, it was identified that RESTful DAS may be optimal for an OLTP database, in terms of performance. To test that theory, we now extend the argument between generic and schema specific DAS for OLTP databases, by exploring how the various approaches perform in an empirical experiment.

A basic experiment was designed to test the performance of executing a single ACID transaction through a DAS. The single transaction was made up of a variable number of constituent actions. The simplest transaction in the experiment had only two constituent actions. The most complex transaction had fifty constituent actions. Varying the level of transaction complexity provided an insight into how the differing DAS implementations coped with increasingly complex transactions.

The experiment executes each transaction one at a time. Doing so removes additional complexities of a full transactional environment, such as: concurrent users; database locking data; differing transaction types. Thus the focus was solely on the performance of the DAS's ability to execute a single transaction. Furthermore, it is likely that if a DAS approach does not perform well for a single ACID transaction, then it will not be viable for a complex OLTP environment.

The transaction designed for this experiment manipulates data in a two table schema. The tables are labeled: *Product*; *StockReplenish*. The schema for *Product* is shown in Table 4.1, and the schema for *StockReplenish* is shown in Table 4.2. The data model is not intended to represent a real world scenario merely it provides tables that allow one to do a transaction that involves multiple inserts and updates.

The transaction itself accepts a variable number of *productIds* for products that require a stock replenish. Varying the number of *productIds*, varies the complexity of the transaction. For each *productId*, the transaction will first insert a record into the *StockReplenish* table, specifying the product that is to be replenished, and the amount of new stock that has been ordered. Then secondly, the transaction will update the *Product* table for the particular product record, and flag that fresh stock has been ordered, by setting the *OrderedNewStock* field to 'Y'. Therefore, the complexity of the transaction is determined by the number of *productIds* that are passed into it. Passing in one *productId* means that the transaction executes two actions. For each additional *productId* that is passed in, an additional two actions are added to the transaction. Thus, complexity increases with the number of *productIds* that are passed in.

Column Name	Data Type	Constraint
ProductId	Number(8)	Not Null
ProductName	Varchar2(20)	
Price	Number(8,2)	
StockLevel	Number(8)	
OrderedNewStock	Char(1)	

Table 4.1: Schema for Product Table

Column Name	Data Type	Constraint
ReplenishId	Number(8)	Not Null
ProductId	Number(8)	
StockQtyOrdered	Number(8)	

Table 4.2: Schema for StockReplenish Table

The experiment was used to measure the time to execute the transaction for the four different DAS scenarios: 1) generic SOAP; 2) schema specific SOAP; 3) generic REST; 4) schema specific REST. Furthermore, a control scenario was created, where no DAS was used, and instead all data communication was done by going directly to the database. The purpose of the control experiment was to provide a baseline for the time it takes to complete the transaction when using a conventional data interaction technique. Each DAS approach can then be compared to the control experiment to determine the impact it has on the transaction time.

For each experiment scenario, the transaction time was measured for a range of increasingly complex transactions that had: 2; 4; 6; 8; 10; 12; 14; 16; 18; 20; 26; 50 actions. To ensure that our measurement of transaction time was reliable, each transaction was executed for each experiment scenario 5000 times. The mean average was then taken to represent the transaction time in a given scenario

To ensure that each transaction could be repeated 5000 times, for each scenario, the *Product* table was pre-populated with $5000 * (TxActions/2)$ records, where $TxActions$ equals the number of actions that are to be executed in the transaction. As the *Product* table was populated, the *OrderedNewStock* column was initially set to 'N' on all records. This is because the *productIds* that are to be executed on, are selected on the basis that *OrderedNewStock* equals 'N'.

As well as measuring transaction time, the experiments also measured bandwidth usage. This measurement was taken to give an insight into how the network is used by each of the DAS approaches. To get this data we measured the amount of TCP data sent and received by the server, using the Linux command `netstat`. The average amount of bytes sent and received for each transaction was recorded over the 5000 cycles.

4.2.1 DAS Implementation

The SOAP based DAS were built in Java. The interface for the generic SOAP service is shown in Table 4.3, this interface has four operations: 1) *execute()* to request a DML statement be executed; 2) *executeQuery()* to request an SQL statement be executed; 3) *commit()* to request that a transaction be committed; 4) *rollback* to request that a transaction be rolled back. To execute the transaction through this DAS, the client first calls the *execute()* operation, passing in the first transaction action as a DML statement, which makes an INSERT on the *StockReplenish* table. Providing the status of the INSERT is successful, the client then calls *execute()* for a second time, on this occasion, passing in an UPDATE statement, to update the *Product* table. These two actions were repeated for every *productId* that was part of the transaction. Then finally, the client calls the *commit()* operation, so that the server could request the DBMS to commit the data changes to the database.

The interface for the schema specific SOAP DAS is shown in Table 4.4, this interface has a single operation: *doTransaction()* that takes a string of *productIds* as a parameter. By passing in a string of *productIds*, the operation has all of the information required to conduct the transaction.

SOAP Genric Interface
execute(String DML, boolean autoCommit)
executeQuery(String SQL, boolean autoCommit)
commit()
rollback()

Table 4.3: Interface for Generic SOAP DAS

SOAP Schema Specific Interface
doTransaction(String ids)

Table 4.4: Interface for Schema Specific SOAP DAS

The RESTful approach uses the HTTP verbs to expose the transaction. To use RESTful services, one must associate request parameters with the HTTP verbs so that the service can understand the client's request. These parameters are passed to the HTTP operations via the URL, or message content. The parameters for the Generic REST DAS are shown in Table 4.5. The client interaction with the DAS is the same as the generic SOAP DAS, only a RESTful interface is used. First the client must make a call to the POST operation, passing in the `createTx` parameter. This tells the service to create a transaction resource for the client. The server will return a URL to the client. The client uses this

URL to execute the transaction. For this transaction, the client will mainly call the PUT operation, setting the `dml_statement` parameter to either an INSERT statement or an UPDATE statement, depending on the action the client wants executing. Once the client has requested all of the transaction actions to be executed, the client will make a final call to the PUT operation, setting the `commit` parameter to true, so that the server will commit the transaction.

Gerenric REST Parameters		
Parameter	**Parameter Data**	**HTTP Verb**
createTx	true	POST
dml_statement	INSERT or UPDATE statement	PUT
sql_statement	SELECT statement	GET
commit	true	PUT
rollback	true	PUT

Table 4.5: Parameters for Generic REST DAS

The parameters for the schema specific REST approach are shown in Table 4.6. For a client to execute a transaction through this an interface, the client calls a POST operation, setting the `ids` parameter to a JSON string that contains an array of *productids* that the transaction is to operate on. Additionally, the `autoCommit` parameter was set to true, to indicate that the server should automatically commit the transaction. In Section 3.2.2 we discussed that one can benefit from a slight break in RESTful principles by having the server automatically run and commit the transaction in the opening POST call. In these experiments `autoCommit` was invariably set to true, allowing for optimal performance. The schema specific REST approach did not need to be stateful, when `autoCommit` was set to true. This was because the server would handle, and coordinate the transaction in the initial POST call, without further client interaction.

Schema Specific REST Parameters		
Parameter	**Parameter Data**	**HTTP Verb**
ids	{ids:[{id:1}, {id:2}, {id:n}]}	POST
autoCommit	true	POST
commit	true	PUT
rollback	true	PUT

Table 4.6: Parameters for Schema Specific REST DAS

4.2.2 Hardware Setup

Two computers were used to conduct these experiments. A computer with a 2.83 GHz Intel Core2 Quad CPU, 3 GB RAM, running Ubuntu 9.04 was used as a client machine. This

machine ran the benchmark experiment application, which simulated the transactions for each experiment scenario. This computer also hosted the web services, which were made available via an Apache Tomcat 7.0.14 server. A computer with a 2.93 GHz Intel Core i7-870, 14 GB RAM, running RedHat Enterprise Linux 6 was used as a database server. The server housed a proprietary DBMS, which was used to store the data for these experiments. Communication between the two computers was established via a crossover cable.

4.3 Results

The results in Figure 4.1 show the average time to complete a transaction, against the number of actions that are in the transaction. The results for all five scenarios are shown in this graph. The results in Figure 4.1 demonstrate that transaction time increases as the number of actions increases, which is what one would expect. Furthermore, Table 4.7 shows the correlation coefficient between transaction time and number of transactions. A correlation coefficient above 0.6 is said to show a strong correlation between two variables. The correlation coefficient indicates a strong correlation between the transaction time and number of transaction actions, for all five approaches.

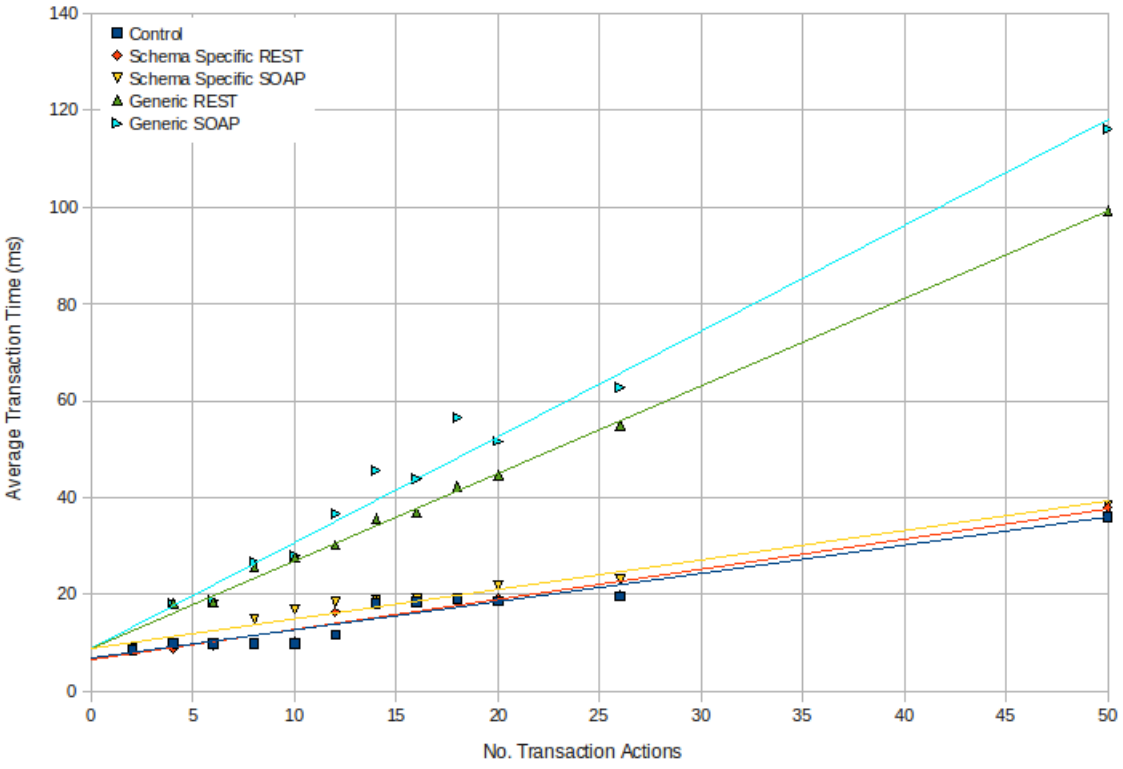


Figure 4.1: Average Transaction Times for Varying Transaction Types for all Experiment Scenarios

Control	SS REST (KB)	SS SOAP (KB)	Generic REST (KB)	Genric SOAP (KB)
0.99	0.99	0.99	1	1

Table 4.7: Correleation Coefficient Between transaction Time and Number of Transaction Actions for each Experiment Scenario

The fact that transaction time increases with the number of transaction actions, is not a revelation. However, what is revealing in these results is the difference in transaction times for the varying approaches, and the rate at which the transaction time increases, for each additional action. The results show that when there are only two actions in the transaction, the DAS, regardless of approach, has a nominal impact on the transaction time compared to the control experiment. As the number of transaction actions increases to four, one can see that the two generic DAS start to place an overhead on the transaction time. Moreover, at four transaction actions, it is approximately two times slower to execute the transaction through the generic DAS. Whereas both schema specific approaches hardly impact the transaction time at all, when compared to the control experiment. The generic approaches continue to have a greater impact on the transaction time, as the number of actions in the transaction increases. By the time of 10 actions, which is a standard transaction in an OLTP environment [26], the generic DAS approaches are approximately three times slower than the control experiment. This trend continues and the generic approach remains around 3 times slower than the control experiment.

Table 4.8 presents the amount of bandwidth used per transaction, for each of the DAS approaches. The results show that for the two schema specific approaches, the amount of bandwidth used per transaction, has no correlation to the number of actions that are in the transaction. The amount of data sent of the network, is approximately the same regardless of the number of transactions. Whereas for the generic DAS, there is a direct correlation between the number of actions in a transaction and the amount of bandwidth used per transaction. As the number of actions increases, the amount of data sent over the network increases.

The bandwidth usage results in Table 4.8, also highlight the differences between the RESTful approaches and the SOAP based approaches. For the schema specific DAS, the RESTful approach uses approximately a third less bandwidth than the SOAP based approach. The RESTful generic DAS also use around a third less bandwidth than the SOAP based approach.

4.3.1 Discussion

The results showed that the generic approach is up to three times slower than the schema specific approach. The theory for why the generic approach is slower than the schema specific approach was discussed in Chapter 3. To recap the generic approach requires the

Tx Actions	SS REST (KB)	SS SOAP (KB)	Generic REST (KB)	Genric SOAP (KB)
2	20.403	30.305	81.603	81.004
4	20.408	30.317	122.403	141.508
6	20.403	30.304	163.208	202.204
8	20.410	30.308	204.009	262.810
10	20.403	30.304	244.807	323.408
12	20.403	30.320	285.609	384.006
14	20.403	30.309	326.409	445.009
16	20.404	30.311	367.216	505.205
18	20.406	30.318	408.006	565.812
20	20.409	30.311	448.805	626.410
Correlation Coef.	0.16	0.36	1	1

Table 4.8: Bandwidth Usage Between Client and DAS, for each Transaction Scenario

client to coordinate the order of the transaction actions and control of the transaction execution alternates between the client and DAS. The client starts by sending an action to the DAS. Control is then passed onto the DAS, so it can execute the action on the database. Then control is handed back to the client, so that the client can send the next transaction action. This switching back and forth, when done over a network, adds significant overhead on the transaction. The results show us that this overhead makes the transaction three times slower.

There is a strong correlation between the number of actions in the transaction and the bandwidth usage, for the generic approach. However, the transactions used in the preliminary experiment, only contains two relatively simple data modification actions, which do not require a large amount of data to be sent to the DAS or to be received from the DAS per transaction. Therefore, the physical network was not a bottleneck, and thus not the cause of the slower performance of the generic DAS. The correlation between the number of transaction actions, and the bandwidth usage, is a result of the fact that a round trip of network communication is required for each action. It is only natural that the more transaction actions there are, the more network round trips there are, and so the more bandwidth is used per transaction.

It is clear from these results that the slow performance of the generic DAS stems from the fact that a round trip of network communication is required for each transaction action. These results confirm the theory in Chapter 3. Now that we know the impact that the generic DAS have on the transaction time, we can now make a comprehensive decision about whether or not to use them for the DAS-CDC approach.

A DAS that makes transactions around three times slower is not feasible for an OLTP environment. OLTP databases are often in high demand [81], which means the DBMS must complete transactions as quickly as possible, in order to cope with the demand. A performance hit of up to three times, means the generic approach will not be feasible in

an OLTP environment. The advantages of the generic approach are more suited to an environment where flexibility is paramount, rather than performance.

The schema specific approach on the other hand, hardly impacts the transaction time. The preliminary results confirm the theory in Chapter 3, that exposing the entire transactional logic as a service operation or resource, allows one to create DAS that is feasible in an OLTP environment. The results in Figure 4.1 show that on occasions, the RESTful schema specific DAS has no impact on the transaction time, and rather, it is quicker than the control experiment. However, the data offers no pattern for why the RESTful DAS are occasionally quicker. One can put it down to possible anomalies in the data. It is possible that on these occasions, a background task, running on the DBMS caused a number of transactions to run a little bit slower, which pushed the mean transaction time up. Moreover, if one were to run these experiments for an infinite number of cycles, the measurements for the control experiment, and RESTful DAS would likely be the same. Alternatively the DAS might be slightly slower, considering the DAS requires one additional round trip of network, that being between the client and DAS.

In both of the schema specific approaches, the bandwidth usage remains around the same, regardless of how many actions are in the transaction. The reason for this is because the client will send a similar sized message to the transaction, regardless of how many actions are in the transaction. Recall that each *productId* that is used in the transaction accounts for two of the transaction's actions. Therefore, the difference between the volume of data used in a 2 action transaction, and a 50 action transaction is 24 *productIds*. A *productId* is at maximum, six characters long. Therefore, at most the difference between the two transactions will be 144 characters. Furthermore, the results in Table 4.8, are an average. For each scenario, the transactions at the begining, will have *productIds* that are one character long, whilst the transactions at the end will be operating on lengthier *productIds*. On a whole though, the bandwidth usage remains roughly the same.

A final observation that can be made from the results is that for both the generic and schema specific DAS, the SOAP implementation performs slightly slower than the RESTful implementation. The reason for this is likely due to the combination of putting extra payload on the network and encoding messages in XML. The SOAP approach uses a third more bandwidth, which is unlikely to make the network a bottleneck, but of course the extra amount of data will take longer to propagate through the network.

The difference between JSON and XML also adds to the additional SOAP service overhead. Nurseitov et al. [82] shows that it is faster to parse a JSON document than a XML document. Nurseitov et al. [82] also show that JSON parsers use less memory and CPU resources when marshalling and unmarshalling messages. Our results also seem to suggest that JSON is the more efficient way to encode data messages.

The reason the SOAP based, schema specific DAS uses approximately a third more bandwidth is due to the fact that the operation data is encoded in a SOAP packet, which is essentially an XML document. XML is more verbose than JSON, and so the XML is the reason for the additional bandwidth usage, seen in the SOAP based DAS.

Finally, we compare our results to previous work. Deng [29] show that it takes three times longer to do single insert statements through a generic DAS, when compared to a conventional ODBC approach. The results presented in this chapter, show that the performance of generic DAS does not improve when doing transactions. We note an overhead that approximately makes generic DAS 3 times slower than conventional ODBC access. On a whole though, it may be difficult to accurately compare out results to previous work, especially in the case of schema specific services. The work in Deng [29], Dogdu [31], Koch et al. [64] measures the speed at which a single action SQL statements can be executed through generic data access services; whereas we show the speed at which multi-action transactions can be executed through both generic and schema specific DAS. Our work confirms the impact of the generic DAS. However, it would be unfair to compare our schema specific DAS to those generic DAS in the literature, because: 1) they use different methodologies; 2) they are being used to execute different operations.

4.4 Summary

The results obtained from the preliminary experiments clearly suggest that a schema specific DAS, built on RESTful principles, is the optimal approach to execute transactions through a DAS. The RESTful approach has a nominal impact on the transaction time, regardless of how many actions were involved in the transaction.

A limitation of the preliminary experiments has been that so far we have only evaluated DASs on a simple transaction. In reality the DAS will need to expose more complex transactions. This could involve the transaction operation needing to receive more than one input parameter. Or it could mean there are more actions in the transaction. Furthermore, a transaction might additionally include query statements as well as data manipulation statements. Additional complexity is also introduced when transactions are run simultaneously, by concurrent users. These preliminary experiments do not consider these scenarios.

Therefore, the next chapter extends the preliminary experiment, to incorporate the additional complexity one can find in an OLTP system. To do this, we benchmark the RESTful, schema specific approach using the Transaction Processing Council's (TPC) TPC-C benchmark. The TPC-C benchmark is aimed to specifically simulate an OLTP environment, allowing one to examine performance of both software and hardware configurations[26]. The extended experiments do not benchmark the performance of the generic DAS, or the SOAP based schema specific approach. Given that the aforementioned approaches are inefficient for a simple transaction, it is unlikely that they will fare any better in an environment with additional complexity.

Chapter 5

Benchmarking RESTful Schema Specific DAS

This chapter evaluates the RESTful schema specific DAS using The TPC-C benchmark [26]. The aim of this experiment is to determine how the DAS performs in a transactional environment, which is more realistic than the experiments conducted in Chapter 4. The TPC-C benchmark is aimed at specifically measuring transactional performance in an OLTP environment, which provides a sufficient examination for the TAAR DAS.

A TPC-C benchmark is a complex implementation. To save development efforts, an open source implementation of the benchmark was obtained from Lussier [71]. Lussier's BenchmarkSQL-2.3.2 implementation is not an accurate implementation of the TPC-C specification. It does not model wait and think times. Wait and think times are not crucial to our experiments, because we are not interested in modelling or measuring human beaviour. In Lussier's TPC-C implementation, a client submits transactions one immediately after the other. BenchmarkSQL-2.3.2 has been implemented using Java. It communicates with the database through JDBC, and uses prepared statements to improve performance.

5.1 TAAR Implementation

A set of services were built to encapsulate the transactions that form the TPC-C benchmark. By using the schema specific methodology, each transaction can be exposed as a resource. We have named the services; TAAR services (Transaction As A Resource services). This section details how we implement a set of TAAR services for the TPC-C benchmark.

The first step to building the TAAR was to determine the transactional logic that each TAAR will expose. We analysed the BenchmarkSQL code to isolate the transactional logic. Five Java methods encapsulate the logic for one of the five 5 TPC-C transactions. Examining these methods allowed us to determine: the transactional logic; what parameters the transaction required to execute; what the transaction will return when it has finished.

Given this information, we could create 5 TAAR services to that encapsulate the logic that was originally in the BenchmarkSQL application.

Each of the five transactions had a RESTful interface, implemented as a Java Servlet. The input to each service was determined by the method parameters of the Java method that the service was replacing. These method parameters become the RESTful requests that get sent in the `POST` call that is used to create and start the transaction. The return data from the transaction's Java method became the service's response. In a way each method became a remote method that was accessed through a RESTful interface.

Each of the five servlet interfaces received the request data in the form of a JSON message. The interface had the responsibility of parsing the input, performing sanity checks on it, and converting it to the correct data type for the transactional logic.

The transactional logic was encapsulated in its own object called a transaction handler. Each transaction had a corresponding transaction handler. After the servlet had parsed the request, it instantiated the correct transaction handler object, and passed in the request data, so the handler could do the transaction processing. The logic in the transaction handler, matched the logic from the Java method that was being replaced in the BenchmarkSQL application.

In order for the handler to do its work, it required an object called a client resource. A client resource is a pooled object that contains a connection to the database, and all of the possible Java prepared statements that a transaction handler may need. Each servlet had access to the client resource pool. Upon on creating the transaction handler, the servlet would obtain a client resource from the pool, and pass it to the transaction handler.

The reason for using a client resource is twofold. Firstly the transaction handler needs a connection in order to do its work. Secondly, in order to keep the TAAR implementation consistent with the BenchmarkSQL implementation, the handler must use Java prepared statements. The transaction handler is transient object, whose life time only lasts for the duration of a single transaction. The prepared statement objects must survive beyond the handler's lifespan in order for them to be utilised. If the prepared statements did not survive, then there would be no point in using them, as they would have to be recreated every time a transaction handler is instantiated, which would be no different to using ordinary statement objects. It was logical to store the connection object in the client resource too, because each prepared statement requires a connection object.

After executing the transaction, the handler returns a response. The transaction servlet packaged this response as a JSON message, which it sends to the client.

To complete the implementation of this system, the BenchmarkSQL application was modified, so that it could communicate with the newly created services. To do this the transaction logic that had previously existed in the five transaction methods was replaced with logic that called the appropriate TAAR service. The method now encoded the parameter data into a JSON document, which gets sent to the appropriate TAAR service, using a HTTP `POST` operation. The method then waited for the response to be returned from the server, which came in the form of a JSON message. This message was parsed,

and the output was displayed according to the TPC-C benchmark specification [26].

It is feasible that this relatively simple approach can be taken to convert any direct to the database applications, into an application that uses TAAR services. We felt that this approach is not overly intrusive, and the majority of the BenchmarkSQL application code remained the same.

Our approach promotes good software design principles, which can followed. After all, a transaction is an absolute piece of logic that requires inputs and produces outputs. Once one has determined what these inputs and outputs are, one can create a service that accepts these inputs and produces the same output. It is possible that the most difficult part of this process is updating multiple applications. However, the basic steps for doing this are detailed below:

1. Isolate transaction logic
2. Determine inputs and outputs
3. Turn the inputs into RESTful request parameters
4. Create the server interfaces to accept and parse transaction requests
5. Create a class to handle the logic for the transaction
6. Link the transaction handler class to the appropriate interface
7. Update the application so that it communicates with the appropriate TAAR services

5.2 Experiment Environment

To gauge the impact TAAR has on the transactional throughput, a control experiment was conducted. In the control experiment, a DAS was not used, and instead the benchmark tool communicated with the database directly through the JDBC.

To get a comprehensive insight on the performance of TAAR and the control experiments, the number of transactions per minute (tmpC) was measured over a range of client loads. Both experiments were run with 10, 30, 50, 70, 90 and 100 clients. For each scenario, the database was pre-populated with ten warehouses. Each warehouse in the TPC-C benchmark occupies approximately one hundred megabytes. Thus, the database was preloaded with one gigabyte for each experiment scenario. Furthermore, for each experiment scenario, there was a ten minute warm up period, before the tpmC measuring started. This period warmed up the database buffers and memory. After the warm up, the tmpC was measured over a one hour period. The transaction mix for every experiment was: *New Order 45%; Payment 43%; Delivery 4%; Stock Level 4%; Order Status 4%.*

The amount of TCP data sent and received by the server was also measured. This information was taken to give an insight into how the network is used by each of the two approaches. To get this data, the average amount of bytes sent and received for

each transaction was recorded over 1000 cycles. Then the total amount of bytes sent and received for each transaction was multiplied by the number of times the transaction was executed in an experiment scenario, so we could obtain the average amount of data sent and received per client scenario, for both the TAAR DAS, and the control experiment.

Two computers were used to conduct these experiments. A computer with a 2.83 GHz Intel Core2 Quad CPU, 3 GB RAM, running Ubuntu 9.04 was used as a client machine. This machine ran the benchmark application, which simulated the client behaviour for each client in the experiment. A computer with a 2.93 GHz Intel Core i7-870, 14 GB RAM, running RedHat Enterprise Linux 6 was used as a server. The server housed a proprietary DBMS, and an Apache Tomcat 7.0.14 server. Communication between the two computers was established via a crossover cable.

5.3 Results and Discussion

The results for the tpmC measurement are shown in Figure 5.1. One can see that for five out of six of the client configurations, the DAS does not have a negative impact on the transaction rate. The DAS and control experiment have around the same performance when 10 and 30 clients are running. However, for the remaining client scenarios the DAS marginally out performs the control experiment. Table 5.1 shows the total number TPC-C transactions per second for both the TAAR and control experiment. The results in Table 5.1 further demonstrate that the TAAR and control experiment have similar performance levels, for 30 or less clients. Then, as the number of clients increases, the TAAR marginally outperforms the control experiment. Ultimately, the results from the experiment show that the TAAR approach is viable in an OLTP environment.

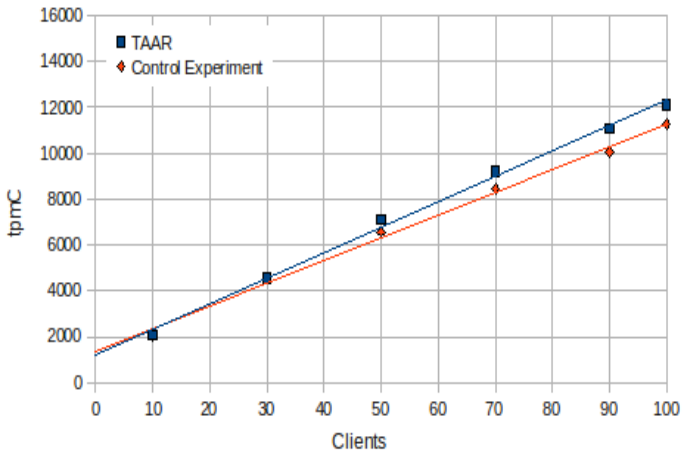


Figure 5.1: Measured tpmC for Transaction as a Resource and Control Experiments for each Client Scenario

Clients	DAS	Control Experiment
10	77	75
30	168	169
50	263	243
70	342	313
90	409	373
100	448	418

Table 5.1: Number of Transctions per Second for Transaction as a Resource and Control Experiments for each Client Scenario

Overall, the viability of the TAAR approach is due to the schema specific implementation. It is a simple, yet effective solution that allows one to reduce the number of network round trips. The approach was further helped due to the fact that the DAS were lightweight enough to be able to run on the same server as the database. We configured a fat server architecture, where the server will serve both the DBMS and the TAAR clients. The benefit of a fat server architecture was that the DAS could communicate with the database, using the server's local buses, rather than via a network. The only data on the network were lightweight JSON messages that the client and DAS used to communicate with each other. The graph in Figure 5.2 shows the total amount of network usage for the various client configurations, for both scenarios. The DAS approach uses approximately three times less bandwidth than the control experiment. Ultimately, the DAS's low bandwidth usage sped up the communication process between the application and server.

The transaction rate in the control experiment starts off at a similar level of performance. When the number of clients is low, the network can handle the traffic between the benchmark tool and the database. As the number of clients increase, the number of transactions increases too, thus, there are more packets on the network. It can be seen in Figure 5.2 that after around 50 concurrent clients, the amount of data on the network, for the control experiment, does not continue to grow at the same rate, as the number of clients increase. This suggests the network is becoming a bottleneck. This correlates with the graph in Figure 5.1 that shows the transaction rate for the control experiment begins to drop off, as more clients are introduced.

Placing the transactional logic on the same sever does come at a cost. The figures in Table 5.2 show the CPU usage for both the control and TAAR experiments, when running the various client configurations. One can see that the DAS approach consumes more CPU resources than the control experiment.

Eventually, with the hardware set up that we were using, the CPU will become a bottleneck for the DAS approach. Moreover, the server will not be able to serve both the DBMS and the DAS without impacting the transaction rate. However, the return of centralised computing can help solve this problem. In computing, there are periodical swings

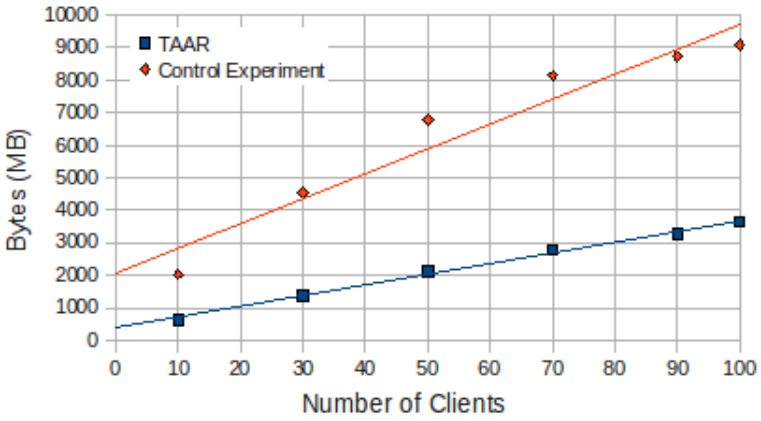


Figure 5.2: Average Amount of Data Sent and Received for Transaction as a Resource and Control Experiments for each Client Scenario

Clients	DAS (%)	Control Experiment (%)
10	19	20
30	24	33
50	31	23
70	37	27
90	43	28
100	49	31

Table 5.2: CPU Ssage for Transaction as a Resources and Control Experiments for each Client Scenario

between using centralised or decentralised IT architectures [38]. A projection in Evaristo et al. [38] suggests centralised computing is favourable again, because centralised solutions are cheaper, reliable and secure. Furthermore, with Moore's law [75] still true today, CPU power is rapidly increasing, and memory is getting faster. The cost of the hardware is no longer an issue [38], thus powerful mainframes and super computers will be open to a wider audience. More recent work by Burleson [15], a renowned industry database expert, also suggests that server consolidation is the current trend in IT. Burleson argues that organisations' systems are bandwidth hungry, and will be so until 2018 where he predicts high-speed satellites will form the back bone of the internet. Until then, organisations' systems will have to be configured to save bandwidth, and "Special K" servers, servers that have 1000 plus CPUs, will enable server consolidation, and thus bandwidth savings [15]. Server consolidation essentially reduces the number of different physical servers. Instead, all of the servers will exist as virtual servers on a mainframe. The servers will be able to communicate via a faster virtual network, rather than the organisation's physical network [69]. The centralisation trend justifies our approach to putting the TAAR on the same server as the DBMS. On a mainframe computer, there will be an abundance of CPU and memory available, enough so that both the DAS and DBMS could run on the same machine, without the CPU quickly leading to a bottleneck.

5.4 Summary

This chapter further evaluated the use of RESTful, schema specific DAS (TAAR) by using the TPC-C benchmark. The aim of the experiment was to ensure that the DAS implementation was viable in an environment that represented an OLTP environment.

The TPC-C benchmark was used to evaluate the TAAR DAS. A control experiment, which used the JDBC to communicate directly with the database, was used also conducted. The control experiment gave us a baseline, which allowed us to determine the impact of the TAAR. The tpmC was measured for both scenarios with 10, 30, 50, 70, 90 and 100 client loads. Overall, the TAAR DAS do not have a detrimental effect on the database's transaction rate. Therefore, the RESTful schema specific approach is the ideal methodology for implementing DAS in an OLTP environment, providing the TAAR services and DBMS are running on the same server. Running both the TAAR services and the DBMS could eventually lead to CPU bottlenecks, and present a scaling problem. However, with a shift towards centralized computing, it is feasible to run both the web services and DBMS on a mainframe, which will have an abundance of CPU power, and memory.

In the following chapters we take the TAAR approach and use it for CDC.

Chapter 6

Benchmarking Change Data Capture

This chapter describes the experimental setup required to evaluate CDC. The chapter begins by discussing the three performance metrics that we shall use to evaluate each mechanism: 1) transaction rate; 2) capture latency; 3) CPU usage. Section 6.2 discusses how the pull CDC mechanisms were implemented, and how the capture tool was implemented. Our novel approach to measuring capture latency is also discussed in this chapter. Section 6.3 details how we use TAAR DAS to do CDC. Finally, in Section 6.4 we discuss the implementation of a target tool which receives and processes the capture data.

6.1 Measurements

The main goals of this evaluation are to answer the following questions:

- what impact does high frequency polling have on the database's transaction rate?
- what impact do the TAAR have on the database's transaction rate?
- how much CPU does each CDC mechanism use?
- how much CDC latency does each CDC system cause, in terms of both mechanical and processing latency?
- Can any of these CDC mechanisms be used for real-time CDC, in accordance to Definition 2?
- In what context can a CDC mechanism be considered for near real-time, in accordance to Definition 4?

To help answer these questions we shall measure three variables: 1) transaction rate; 2) capture latency; 3) CPU usage. In this section we take each of these variables and discuss how we measured them.

6.1.1 Transaction Rate

We measure the transaction rate so that we can determine the affect each CDC mechanism has on the database's throughput. A high transactional throughput is critical in an OLTP database. Theory suggests that pull CDC will use too much of the database's resources if one uses it for real-time CDC. Moreover, high frequency polling will cause contention in the database, which will have a negative effect on the database's transactional throughput. The theory also suggests that push CDC is less likely to have a negative impact on the database's transactional throughput. To test this theory we shall measure the impact each CDC mechanism has on transaction rate.

To measure the transaction rate, we shall continue to use the TPC-C benchmark application that we used for the preliminary experiments in chapter 5. The CDC experiments will be more thorough than the preliminary experiments. We shall continue to use the 10 warehouse configuration, which will represent the database under normal conditions.

We also measure the transaction rate under more extreme database conditions. The unit of concurrency in the TPC-C benchmark is the warehouse. Most transactions target a single warehouse, and two simultaneous transactions that target the same warehouse are likely to conflict, by competing for locks on a given warehouse's data. The more clients there are, and the fewer warehouses there are, the more likely two or more transactions will conflict. However, if there are plenty of warehouses, transactions will be less likely to conflict, and the system would have ample concurrency. With more concurrency, a larger number of transactions will be able to execute simultaneously. However, computer systems and hardware have upper limits, and the more simultaneous transactions there are, the more likely these upper limits will be reached. In the TPC-C experiments, a high number of warehouses will lead to disk I/O limits being reached. With more warehouses, the chances of two transactions targeting the same warehouse decreases, and therefore there will be more disk I/O as the hard disk seeks the appropriate warehouse for the transaction. So we measure the database under two extreme conditions; 1) 1 warehouse configuration, which will see the database limited by contention; 2) 20 warehouse configuration, which in trial experiments, without CDC showed the database become I/O bound. Measuring the impact CDC has on the transaction rate under extreme conditions should help to expose any differences between pull and push CDC architectures. It is expected that push CDC will still perform well in a database that has poor performance.

6.1.2 Capture Latency

To get an understanding of the real-time viability of CDC technologies we shall measure the time it takes to do CDC. More specifically, we shall have to measure the amount of capture latency in each CDC system.

Capture latency does not have a standardised definition in the field. A time measurement requires two points of reference: a start time and an end time. For capture latency, these two reference points have not been defined. Raitto [92] measures capture latency for

Oracle log CDC, and Oracle trigger CDC. However, Raitto does not detail how capture latency is measured. One is left to assume that the start time is the point at which a change is made. Whilst the end time is the point at which a change appears in an oracle change table. We argue that Raitto measures identification latency rather than capture latency, because the changes have not been captured from the oracle change tables. In this thesis we have argued that records are not captured until they have been retrieved from the database. Oracle [85] and HVR Software [52] both claim that log CDC has a sub second capture latency. However, neither provides details on how they make this measurement, or what their definition of capture latency is.

Bouzeghoub [12] provides a literature survey and presents various definitions regarding data freshness, whilst exploring how the different metrics that are used to represent data freshness and latency. Bouzeghoub's research found that there are four types of data freshness metrics. Table 6.1 shows Bouzeghoub's latency metric definitions.

Metric	**Definition**
Currency	The time difference between extracting the source data, and it being delivered to the target.
Obsolescence	The number of transactions executed on a source since the last data extraction time.
Freshness Rate	The number of tuples that have been updated since the last extraction point.
Timeliness	The time since the source data was last updated.

Table 6.1: Bouzeghoub's Latency Metric Definitions

Bouzeghoub's definitions are either irrelevant or too broad to define capture latency in the context of a CDC system. *Currency* is not suitable because the time measurement begins after the change data has been extracted. This is a metric that would be used when measuring a change delivery system. *Obsolescence* and *Freshness Rate* are not applicable to us because they are measuring ratios, not time. Finally, *Timeliness* does not concern the time measurement that we are interested in. Timeliness is more useful when one requires data about how often one's database is updated. This might be a more useful metric in a real world environment. Practitioners could use this metric to determine how often one should poll the database for changes.

The likely reason that no standard metric exists for measuring capture latency is because in a transaction environment it is a difficult to precisely and consistently measure how long it takes to capture a change, once the change has been made [12]. Therefore, estimations are often used to represent this time, as Theodoratos and Bouzeghoub do in [103]. Here they use the *Currency* metric from Table 6.1 to represent the time it takes to refresh a data warehouse view, with fresh operational data. However, this includes the extra latency of loading a warehouse, where we are concerned only with capturing the changes.

Making a consistent measurement for capture latency is not straight forward in an OTLP environment. At first glance it would be intuitive to use Equation 6.1, where Tc is the time when the change was captured, and Tm is the time when the change was made. Tm could be made by recording a timestamp with the change, and Tc could be made when the change is captured. However, in an OLTP environment, the point at which Tm is taken, is not consistent for every transaction action. OLTP transactions are made up of multiple actions which are executed chronologically. Due to transaction isolation, these changes will not be seen in the database until the transaction commits, but the timestamp is still created at the time the action occurs. If we were to take Tm at the time when a change is made, then the capture latency for that change will include the time it takes the transaction to do any subsequent transaction actions. This means we will get inconsistent measurements for capture latency.

$$Cl = Tc - Tm \tag{6.1}$$

The following example considers this point further. Take a transaction: Tx1, which is made of the following actions: insert; select; insert; update; commit. Table 6.2 shows the chronological execution of these actions and the time at which all of these actions happen. The capture latency will be different for each change: Equation 6.2 shows the capture latency for Ch1; Equation 6.3 shows the capture latency for Ch2; Equation 6.4 shows the capture latency for Ch3. One can see that the capture latency will be different for each change, because the value of Tm from Equation 6.1 differs for each change. When using Tm for the start time, capture latency becomes dependent upon where the action is situated in the transaction; the earlier an action happens, the more capture latency there will be. Using Equation 6.1 for capture latency will lead to inconsistent capture latency measurements.

Source	Action	Time Action Finishes	Change Number
Tx1	insert	T1	Ch1
Tx1	select	T2	
Tx1	insert	T3	Ch2
Tx1	update	T4	Ch3
Tx1	commit	T5	
CDC Mechanism	Capture Changes	T6	

Table 6.2: Timeline of a Transaction with Four Actions

$$Ch1 = T6 - T1 \tag{6.2}$$

$$Ch2 = T6 - T3 \tag{6.3}$$

$$Ch3 = T6 - T4 \tag{6.4}$$

A fairer measurement for capture latency's start time is the time when the transaction commits. When the transaction is committed all changes made in the transaction will become visible to other database users. Consequently, the capture time will be the same for all changes that were made in the transaction. Equation 6.5 expresses the capture latency for a change when the start time is taken at the commit time, $Tcom$.

$$Cl = Tc - Tcom \tag{6.5}$$

Measuring the commit time in pull CDC architectures is less than trivial, as DBMSs do not offer the ability to do this. This led us to create a novel technique to measure commit time, and subsequently the capture latency, in pull CDC. We discuss this further in Section 6.2.1. Measuring commit time is simpler for push CDC, which we discuss in Section 6.3.1.

The other variable required to calculate capture latency, is the time at which a change is captured. The capture time is idiosyncratic to the CDC architecture, and implementation. Furthermore, it is debatable as to what constitutes as a captured record. We have this discussion for pull and push CDC architectures in Section 6.2.1 and Section 6.3.1 respectively.

6.1.3 CPU Usage

Pareek's [88] empirical study on CDC mechanisms measured CPU usage. We too feel that this is an important measurement. CDC is likely to be a CPU intensive operation; therefore it is worth determining how much additional CPU usage CDC requires.

The measurement is particularly important for determining the cost of TAAR CDC. In the preliminary experiments, where the TAAR did not implement CDC, we saw that the TAAR places an overhead on CPU usage, when compared to a standard direct to the database approach. We need to determine whether the additional CDC workload, causes the TAAR approach to become too CPU intensive.

6.2 Pull CDC Implementation

6.2.1 Capture Latency

We have previously mentioned that measuring the commit time is not trivial in a DBMS. Typically DBMSs do not record the time at which tuples are committed to the database. The only DBMS that comes close to doing this is Oracle. Oracle records an SCN (System Change Number) number with each DML operation, which is essentially a metadata identifier for the transaction. The SCN can be used to calculate the time at which the change was made, but it is only accurate to ±3 minutes. We require millisecond precision to measure capture latency.

An approach to recording the commit time is proposed by Myers in [76]. Myers's approach requires each record in every table to have an additional commit time column,

which is set as null, when a record is updated or inserted. A self programmed DBMS background process, then continually seeks records that have a commit time column that equals null. When the background processes encounters a null, it replaces the null with the current system time. There are numerous problems with this technique: 1) there will likely be a small amount of latency before the null value gets replaced; 2) the background process will consume DBMS resources that could affect the transaction rate; 3) in a database with a heavy workload, the capture tool might capture the change before the null value has been replaced, meaning we lose the commit time measurement.

An alternative approach to measuring the commit time is to create a materialized view of each table that gets changed. The materialised view will only get populated with fresh change data after the change has been committed. A trigger could be setup on the materialised view to fire when new records are placed in to the view. The trigger could write the commit time, to a timestamp column in the materialised view for each change. However there will likely be a delay in writing the commit time. Furthermore, the extra operations could place an overhead on a transaction, by giving the DBMS too much work to do.

With no feasible mechanism for measuring commit time, we implemented a technique that is tailored for these experiments, but can be used to measure commit time in any OLTP environment. Our approach tags each transaction with a unique *transactionId*. When the transaction makes a change to the database, the change is tagged with the *transactionId*. To store the *transactionId* an additional column was added to the tables in the TPC-C schema. When the transaction commits, a commit time is associated with its *transactionId*. The benchmark application writes the *transactionId* and the commit time to a key value store. When a change is captured, the *transactionId* for that change is also captured. The capture tool will record a capture time for each *transactionId*. The capture tool then writes the *transactionId* and capture time to a separate key value store. Once the experiment scenario finishes, a capture time calculator iterates over the two key values stores, using the *transactionId* to match a capture time to a commit time. The capture time and commit times are then plugged into Equation 6.5, so that we could obtain the capture latency for a change.

An issue with this approach is that the commit time and capture time are recorded on separate computers. The commit time is recorded on the computer that is running the BenchmarkSQL application, whereas the capture time is recorded on the database server, because the capture tool is running on the same computer as the database. To ensure that capture latency could be accurately calculated, the two clocks had to be kept in synchronization. We use the Network Time Protocol (NTP) to synchronize the two clocks. The computer running the BenchmarkSQL application acted as an NTP server, and the database server was a client to the NTP server. Using NTP, the time offset between the two clocks was sub-millisecond, which is accurate enough for our evaluation.

6.2.2 Pull CDC Mechanism Implementation

6.2.2.1 Timestamp CDC

Timestamp CDC required the tables in the TPC-C schema to be extended, so that they include a timestamp column, which is used to store the time at which a record is inserted or updated by a DML operation. The timestamp column was set up on the eight tables whose tuples are modified by the TPC-C transactions. Each timestamp column, stores a timestamp data type, with millisecond precision. Every DML statement that the BenchmarkSQL application generates in the transaction logic was modified to populate the timestamp column with the DBMS system time.

The capture tool uses the timestamp column to identify fresh changes. More specifically, a query will use the timestamp column to find the latest changes. Figure 6.1 shows a generic query for timestamp polling. The query will select all of the fields from the table, because timestamp CDC has no mechanism to identify which field specific field was changed. The `WHERE` clause instructs the query processor to retrieve changes where the value in the `change_time` column, the timestamp column, is greater than a parameter. The parameter to this query is the time of the eldest timestamp from tuples returned in the previous poll operation. This ensures that the query only returns changes that have been made since the last poll operation; the freshest change data.

```
SELECT <table fields>
FROM <table>
WHERE change_time > <previous_timestamp>
ORDER BY change_time
```

Figure 6.1: Generic Query used in Timestamp CDC Polling

To help improve the efficiency of the capture tool's queries, an index was applied to each of the timestamp columns. The index used, was a standard B-tree index.

6.2.2.2 Trigger CDC

Trigger CDC required us to create triggers that would fire whenever a DML operation is executed on an operational table. We identified all of the DML operations that were applied to the tables by the TPC-C transactions and created a trigger for each table where appropriate. We also created a staging table for each operational table, which the trigger populated with change data. The triggers essentially copied the fresh data into the staging table after an INSERT or UPDATE was applied to the source table. The staging tables contained the same fields as the operational tables and an additional field to store a `change_id`.

The capture tool creates a poller for each of the staging tables. The poller queries these staging tables, and uses the `change_id` to identify new changes. Figure 6.2 shows a generic query that a trigger CDC poller will issue against the database. The query selects

all of the operational table's fields, and the `change_id`. The `WHERE` clause instructs the query processor to select tuples where the `change_id` is greater than a parameter. This parameter is the highest `change_id` found in the previous set change results. Because the `change_id` values are sequential, the query will only return changes that the poller has not previously extracted.

```
SELECT <table fields>, change_id
FROM <table>
WHERE change_id > <previous_id> ORDER BY change_id
```

Figure 6.2: Generic Query used Trigger CDC Polling

6.2.2.3 DBMS Log Scanning CDC

The commercial DBMS used for these experiments, offers a DBMS log CDC technique. To set up DBMS log CDC, we created a capture process for each of the eight tables that are involved in the TPC-C transactions. Part of log CDC initialisation required us to create a set of change tables; each operational table is associated with a change table. Each change table contains a replica of the columns that are found in its corresponding operational table, as well as metadata such as: a status to identify the type of DML operation; a transaction identifier; a unique `row_id`; a timestamp; the name of the user who made the change. The change tables are populated by a DBMS process, which will scan the DBMS's redo logs, to seek out fresh DML statements. Upon discovering a DML statement, the process will parse the log entry, before inserting a change record into the change table that corresponds to the operational table that the DML statement was applied to.

The capture tool creates a poller for each of the capture tables. The poller queries these capture tables to identify the latest changes. Figure 6.3 shows a generic query that a DBMS log CDC poller will issue against the database. The query selects all fields that are in the operational table, and the change table's `row_id`. The `WHERE` clause instructs the query processor to select tuples where the `row_id` is greater than a parameter. This parameter is the highest `row_id` found in the previous set change results. Because the `row_id` values are sequential, the query will only return changes that the poller has not previously extracted.

```
SELECT <table fields>, row_id
FROM <table> WHERE row_id > <previous_id> ORDER BY row_id
```

Figure 6.3: Generic Query used in DBMS Log CDC Polling

6.2.3 Capture Tool

For the purpose of doing pull CDC, a Java based capture tool was built. The job of the capture tool is to poll the relevant change resource, to extract the data changes from it. It was possible to configure the capture tool to poll both the staging areas used for trigger CDC, and DBMS log CDC, or the operational data tables, when timestamp CDC was being evaluated.

We examined the TPC-C transactions to determine which operational tables were having changes applied to them. These tables were: 1) new_order; 2) oorder; 3)order_line; 4) stock; 5) warehouse; 6) district; 7) customer; 8) history. In each pull CDC scenario, the capture tool will poll each of these 8 tables, or the corresponding staging table, when applicable.

The capture tool polls the 8 resources, by spawning a poller thread for each resource. The poller thread periodically polls the table that it has been set up to monitor. The frequency of the poll operation is determined by a poll interval setting. The poll operation is a SQL query, which is implemented as a prepared statement. Each poller has its own connection to the database that it will use to execute the query. The syntax of the query for each type of CDC was discussed in Section 6.2.2.1, Section 6.2.2.2 and Section 6.2.2.3, for timestamp CDC, trigger CDC, and DBMS log CDC respectively.

The response from each poll query is a set of results. The result set contains the tuples that match the polling query. To fully capture the changes the result set has to be processed, because at this point the result set data structure is in the capture tool's memory context. The result set data structure cannot be passed between applications, so in order to send the changes to a target one has to process to the result set to actually obtain the change tuples. The same is true of all programming languages, the results in the initial result set container have to be read before they are captured. If the result set is closed before one can process it, the tuples in the result set will no longer be in the application context, and thus the tuples belonging to that resultset will not have been captured.

The processing of the result set is done by a change sender. Change senders are threads that get spawned during the creation of the capture tool, for the purpose of processing change result sets. The alternative was to process the result sets in the poller thread. A result set could contain a large number of results, and processing it can take time. If the poller thread processes the result sets then it could delay subsequent poll operations. The poller threads and the change sender threads share a queue. Poller threads place result sets on to the queue, and the change senders take result sets off the queue and process them. A change sender processes a result set by serialising it into a JSON message.

The capture time for a change is taken when the change sender serialises the result set into JSON. We take the capture time here, because this is the point at which the change is out of the database, no longer in a result set object, and in a format which will allow the changes to be shared with target systems. Alternative options for taking the capture time include: the point at which the poller thread queries the database, but this would

mean the capture time is taken before the changes are read from the database; the point at which the poller thread's query finishes, but as previously explained at this point changes are in a data structure that is usable only by the capture tool. We feel it is fair to take the capture time during the serialisation process because, all changes have to be serialised before the target can use them. Furthermore, no matter which programming language was used to implement the capture tool, some form of result set serialisation would have to be done, before the changes could be used by a target system. Essentially changes are captured, once they are out of the database, and into a format that allows them to be shared with target systems.

6.3 Push CDC

Chapter 5, Section 5.1 discussed how the TAAR were implemented for the TPC-C benchmark. In this section we discuss how CDC logic was added to this implementation.

As previously mentioned in Section 5.1, the logic for each TPC-C transaction is implemented in a separate class; a transaction handler. The transaction handler contains a method that encapsulates the transactional logic. If one of those actions executes a DML statement on the database, then the changes it makes need to be captured. In TAAR CDC, it is the TAAR that does the capturing, not some capture tool. The most logical place to capture a change is immediately after the action that made the change was executed.

Chapter 3, Section 3.1.2 explains the theory and conceptual logic of service based CDC. Each transaction has a change message that gets built as the transaction applies changes to the database. After a given change, details regarding the change are appended to the change message. Providing that the transaction successfully commits, the change message is then sent to target. We implemented that logic into our TAAR services for the TPC-C experiment.

To do this we modified each transaction handler, so that after the handler executes a DML operation on the database, it creates a change message. In the practical implementation this message became a change object. A change object is a custom built object which stores details about a DML change. A change object stores the following data fields:

- **action:** The type of DML operation that the action executed. The values will either be: `INSERT`; `UPDATE`; `DELETE`.
- **table:** The name of the table that the DML operation was applied to.
- **where:** The where attribute is an optional attribute, to be used when the DML operation is an `UPDATE`. The attribute is a 2D array that stores the `WHERE` clauses that were used to identify the tuple or tuples that were to be updated. Each element of the array has the following attributes: `column`, to store the name of the column that is to be used in the comparison; `operation`, either $=,<,>,\leq,\geq$ that was used

in the `WHERE` comparison; `value` the value to which `column` is being compared to. Storing the where data is vital, so that the target will able to use this information to deduce which source tuple(s) have been updated.

- **change:** This attribute is a 2D array to store the changes that were made by the action. Each element of the array has three attributes: `column`, the name of the column where the data has been changed; `value`, the value that the column was set to; `operation`, an optional attribute that can provide further information about how the data was changed, for example "plus"" would tell the target that the field was changed by adding the `value` to the current value, rather than the column being set to the value.

The change object gets created after the change is made. The data that gets added to the change object's change array comes from the variables that were used to populate the prepared statement's change data. Similarly, the data from the `WHERE` clause in the prepared statement, if there was a `WHERE` clause, is used to populate the change object's where array.

Once the change object is populated with all of the data about the change, it is added to a transaction change object. The transaction change object contains the following fields:

- **txName:** This attribute stores the name transaction. This information is stored, as it might be useful for the target to know which business transaction caused the change. This could be particularly useful, if more than one transaction updates a table; the name would tell the target which transaction was responsible for the change.
- **changeList:** This is a list of change objects that are associated with the transaction.
- **commitTime:** An attribute that stores the time at which the transaction was committed.

Each transaction creates a transaction change object, which it maintains throughout the transaction. As the transaction executes its constituent actions, the change objects that are created after each action are added to the transaction change object. After all of the transaction's constituent actions have been executed, the transaction will attempt to commit the data. Upon confirmation of a successful commit, the TAAR records the commit time, using a timestamp, with millisecond precision. The transaction change object's commit time is then set to equal the recorded commit time. Taking the commit time at this point is consistent with how the commit time is taken in the pull CDC approach.

The TAAR then places the transaction change object onto a change queue. The TAAR services share a single queue change queue. The change queue is similar to the change queue that we implemented for the capture tool for pull CDC. Only this time the queue holds transaction objects, rather than result sets

Similar to the capture tool, we created a change sender class that takes change objects off of the change queue and processes them. The change sender thread takes transaction objects off the queue, and processes them, by serialising its data into a JSON document. Changes that belong to a transaction are packaged into the same JSON document, thus maintaining transaction consistency.

6.3.1 Capture Latency

The commit time was recorded immediately after the transaction had successfully committed its actions, which is same point where commit time was measured in the pull CDC experiments. For the pull CDC approach one needed to tag the commit time with a *transactionId*, so that one could later map it to the time the change was captured. However, in the TAAR approach, one did not need to tag changes with a *transactionId*, because changes were captured in the same application context that committed them. Moreover, the commit time and the capture time were recorded by the same application.

The capture time for a change is recorded when the change sender processes the transaction object's array of change objects. When the change sender converts the change object into a JSON string, it will take a system timestamp with millisecond precision. This timestamp will represent the time that the TAAR captures the change. We take the capture time here because this is where the TAAR truly captures the change data. When the change data is in the change object it is still technically in the TAAR's application memory context, and cannot be used in another system.

6.4 Target Tool

The target system was the same for all CDC mechanisms. The capture tool or the TAAR sent each change to the target system. As previously discussed all changes were encoded in a JSON message. The target used for these experiments did a minimal amount of work. Because we are only interested in capture times, we did not implement a fully fledged target system that did anything useful with the data, such as transform it and load it into a warehouse. Instead, all the target did was parse the received JSON documents to obtain the capture time for each record and write it to the capture latency file. Or in the case of TAAR CDC, it recorded both the commit time and the capture time, as both were sent in the JSON message.

6.5 Summary

This chapter described the experimental method for evaluating CDC technologies. The experiment evaluates CDC on three key variables: 1) transaction rate; 2) capture latency; 3) CPU usage.

Our experiment is based around the TPC-C benchmark, which is an industry recognised benchmark for measuring the transactional performance on an OLTP database. By

using this benchmark we are able to obtain empirical results regarding the impact CDC technologies have on a database's transactional throughput. The optimal real-time CDC mechanism will have the least impact on the transaction rate. We evaluate the CDC technologies with a range of TPC-C configurations, by varying the number of warehouses and the number of concurrent clients. In total there are eighteen experiment scenarios for each of the four CDC mechanisms. The TPC-C configurations include a 1 warehouse scenario, a 10 warehouse scenario and a 20 warehouse scenario. For each warehouse scenario, there are experiments for 10 clients, 30 clients, 50 clients, 70 clients, 90 clients and 100 clients. These experiment scenarios vary the load on the database, thus allowing us to gain a comprehensive insight on the performance of each CDC mechanism.

We extended the TPC-C benchmark in order to measure the amount of capture latency that each CDC mechanism is responsible for. None of the previous work in the literature provides a definition for capture latency. To fill the gap, this chapter provided a definition for capture latency. Capture latency is the amount of time between the point where a change is committed to a database, and the point where the change is captured. The least invasive approach to measuring these two points was to tag each change with a commit time and a capture time. To make the capture latency measurement was consistent for all four CDC mechanisms, we only record the capture time measurement, once the change has been fully captured, and is ready to be sent to the target. Although we used the TPC-C benchmark for our evaluation, our capture latency definition and measuring technique is independent of the TPC-C benchmark, and can potentially be used in other CDC evaluation systems.

Our CDC evaluation also measures the amount of CPU that each CDC mechanism uses. CDC is a CPU intensive task and it is worth determining how much CPU each mechanism requires, as this could have implications on whether a CDC mechanism is viable. Furthermore, our preliminary experiments in Chapter 5 determined that TAARs are CPU intensive. We now need to establish whether the additional CDC workload makes TAAR non-viable due to overloading the CPU.

Finally, this chapter detailed the implementation of the various CDC techniques. The TAAR CDC mechanism uses the TAAR Java servlets, which were developed for the preliminary experiments in Chapter 5. The TAAR have been modified to include the CDC logic. For the pull CDC mechanisms, a Java based polling tool was created, which polls the relevant change resources to extract fresh change data. For all of the experiments, the change messages are built as JSON strings. These JSON strings get sent to a target tool, which is also Java based.

Chapter 7

Change Data Capture Benchmark Results

This chapter presents the results that were obtained from the TPC-C CDC experiments. The aim of the experiments is to provide data on key performance metrics: transaction rate, capture latency and CPU usage, for each of the CDC mechanisms. The data obtained from these experiments will be used to underpin our theory on the performance of CDC mechanisms. Ultimately the data will allow us to test our hypothesis that push CDC will be capable of real-time CDC, whereas high frequency polling in pull CDC architectures will cause a significant impact on the database's transaction, thus making the architecture unsuitable for real-time CDC.

This Chapter starts by presenting the transaction performance data, followed by the capture latency data, and then finally the CPU usage results. The interpretation of the results follows in Chapter 8.

7.1 Transaction Rate

This section presents the results on the database's transactional performance when each of the different CDC scenarios were set up to capture changes that were generated by the TPC-C benchmark. Transactional throughput is measured in tpmC, which is a measure of the number of TPC-C new order transactions that happen every minute. This section also presents the running tpmC average, so that one can gain insight into how the transaction rate performed over the course of the experiment period. Because the tpmC only considers the number of new order transactions per minute, we also present results for the total number of TPC-C transactions per second, which includes the other four TPC-C transactions.

The optimal CDC mechanism, in terms of transaction rate, will be the one that has the least significant impact on the transaction rate.

7.1.1 Normal Conditions

We start by presenting the transaction rate results for when the database is configured with 10 warehouses.

Figure 7.1 presents the tpmC results for each of the experiment scenarios at the 10 warehouse configuration. Table 7.1 presents the results for the total number of transactions per second for each experiment scenario at the 10 warehouse configuration.

One can see that the TAAR and control experiments exhibit similar performance levels, that were noted and discussed in Chapter 5. What is interesting to note, is that even with CDC enabled, the TAAR approach still slightly outperforms the control experiment.

We observed, that compared to the control experiment, timestamp CDC had a nominal impact on the transaction performance at the lower client numbers (50 or less). At the higher numbers of clients, the timestamp approach placed a significant reduced the transaction rate by of 52%, 65% and 70% for the 70, 90 and 100 client scenarios respectively. Both Trigger CDC and DBMS log CDC placed a nominal overhead on the transaction rate at the 10 client scenario. However, from 10 clients onwards we observed significant overheads on the transaction rate for both CDC techniques. This suggests that the triggers and log scanning mechanisms carry severe performance penalties.

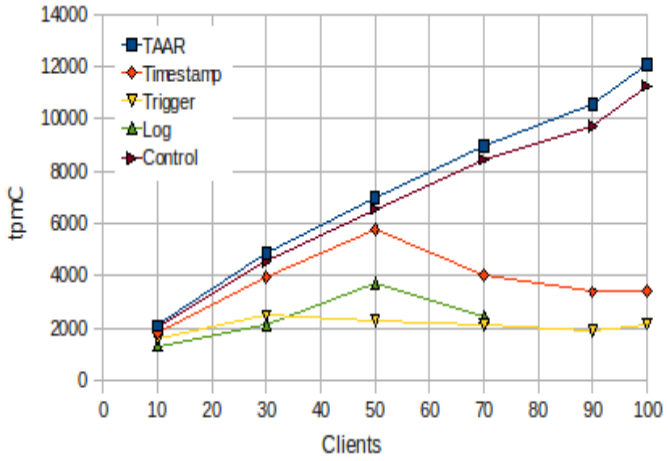


Figure 7.1: tpmC for each CDC Mechanism and Control Experiment for Various Client Scenarios; 10 Warehouses

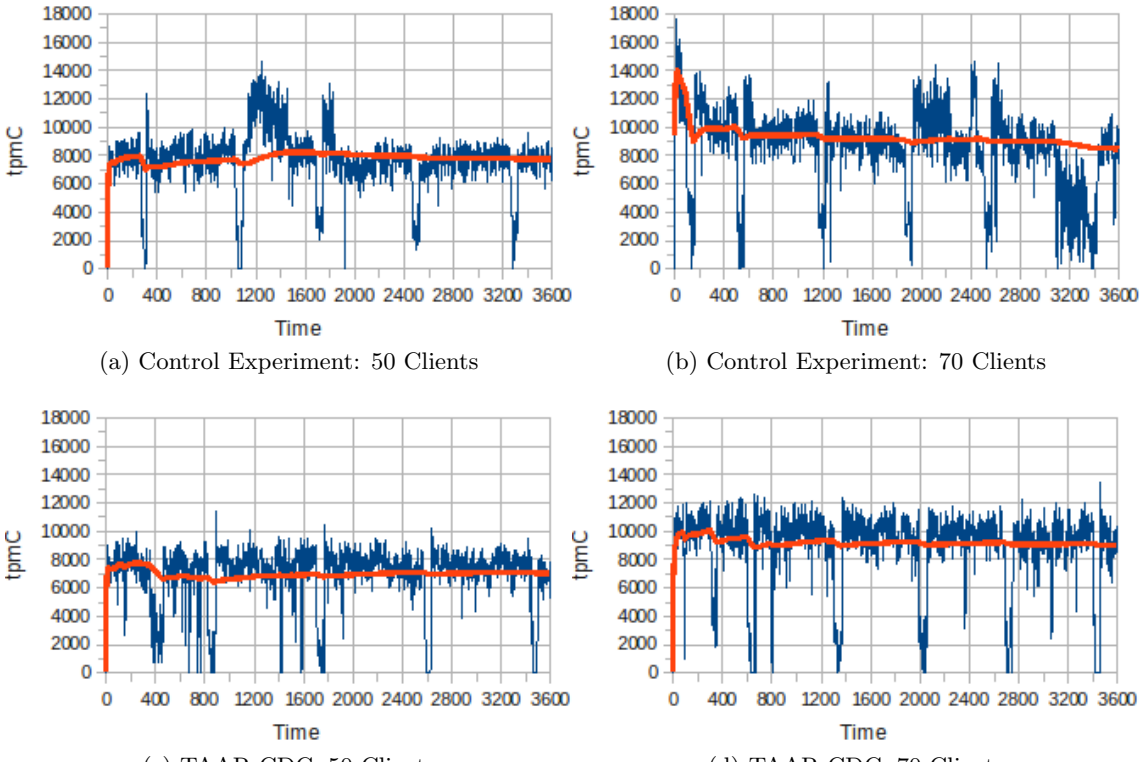
Clients	TAAR	Timestamp	Trigger	DBMS Log	Control Experiment
10	77	67	59	49	75
30	179	146	91	78	169
50	259	215	85	137	243
70	332	150	79	90	313
90	391	125	70	-	359
100	448	127	73	-	418

Table 7.1: Total Number of Transactions per Second; 10 Warehouses

Figure 7.2 plots the tpmC at two second intervals for each CDC scenario and the control experiment, for the 50 and 70 client scenarios. The red line indicates the running tpmC average and the blue line indicates the actual tpmC value. We show the plots for the 50 and 70 client scenarios because the show a key transition in transactional performance as the number of concurrent clients increases.

Each of the tpmC running average graphs exhibit regular spikes of low performance. For example, in Figure 7.2a we observe low spikes at approximately 300 seconds, 1050 seconds, 1700 seconds, 2500 seconds and 3300 seconds. We have identified that these low spikes coincide with database redo log switches. Common behaviour for an OTLP database is to archive redo log files when one redo log becomes full [45]. In the process of switching redo logs, the filled up log is archived, and excessive I/O can occur at this point [16], which will temporarily cause low performance.

The running tpmC performance graphs in Figure 7.2 give an insight into how the transaction performance was affected over the course of the each experiment scenario. For the pull CDC techniques, especially at the 70 client scenario (Figures 7.2f, 7.2h and 7.2j), one can see that the transaction performance started off in a similar fashion to the control experiments, but after a period of time, the tpmC becomes inconsistent, and fluctuates between periods of high and low rates.



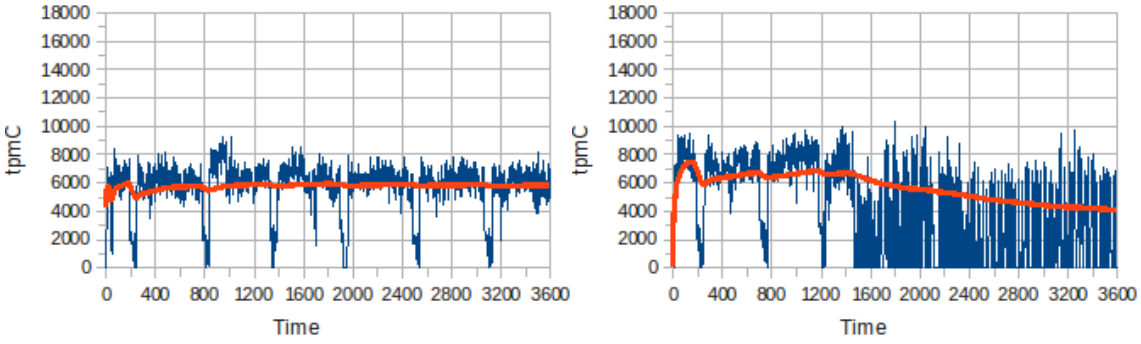
(a) Control Experiment: 50 Clients

(b) Control Experiment: 70 Clients

(c) TAAR CDC: 50 Clients

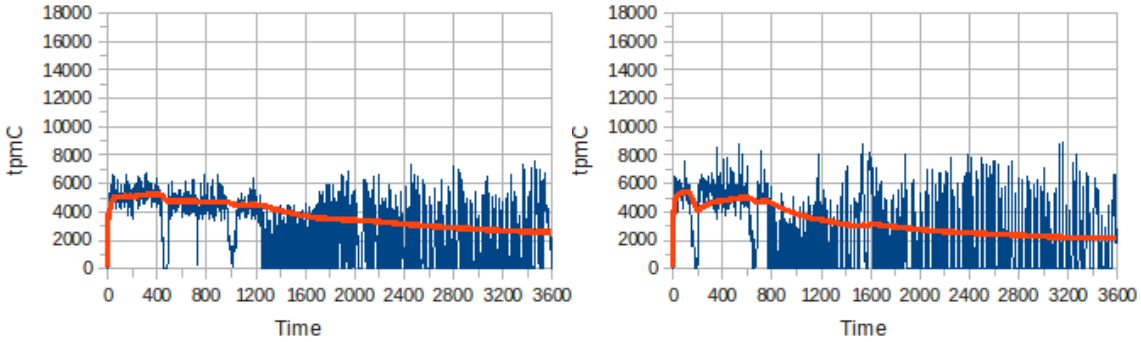
(d) TAAR CDC: 70 Clients

Figure 7.2: tpmC at 2 Second Intervals; 10 Warehouses



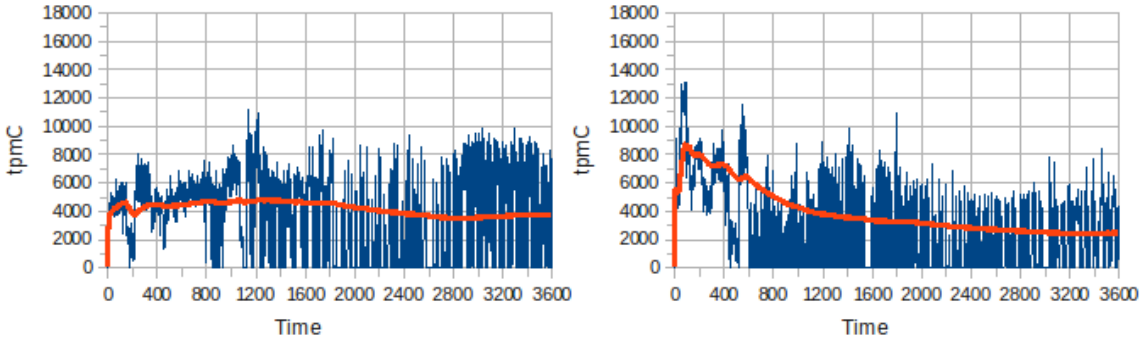
(e) Timestamp CDC: 50 Clients

(f) Timestamp CDC: 70 Clients



(g) Trigger CDC: 50 Clients

(h) Trigger CDC: 70 Clients



(i) DBMS Log CDC: 50 Clients

(j) DBMS CDC: 70 Clients

Figure 7.2: tpmC at 2 Second Intervals; 10 Warehouses

7.1.2 Extreme Conditions

7.1.2.1 1 Warehouse Configuration

The 1 warehouse configuration causes high levels of contention in the database, which will ultimately cause database lock contention, thus limiting transaction concurrency.

Figure 7.3 shows the tpmC performance for each CDC mechanism and the control experiment at the 1 warehouse scenario. For all of the experiment scenarios the transaction rate levels off at from 30 clients onwards. This suggests that at database becomes bound by contention and the transaction rate does not increase when more clients are added. The results show that the three pull CDC mechanisms exhibit nominal impact on the transaction rate compared to the control experiment. The TAAR again performs better

than any of the other scenarios.

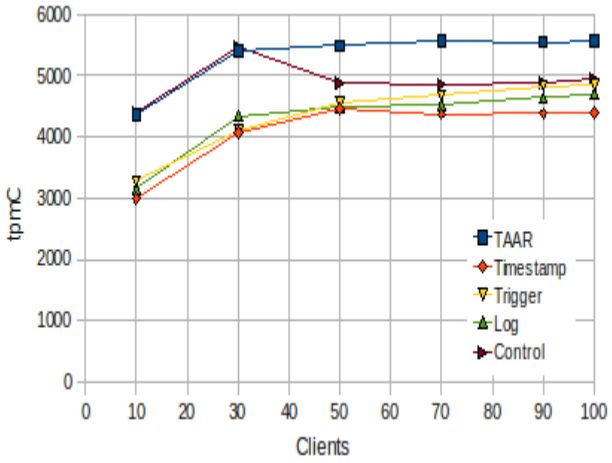


Figure 7.3: tpmC for each CDC Mechanism and Control Experiment for Various Client Scenarios; 1 Warehouse

7.1.2.2 20 Warehouse Configuration

The 20 warehouse configuration provides the database with ample concurrency, and transactions are less likely to suffer from lock contention. However, with more warehouses the database has more data in it, and thus will have to do more disk seeking to fulfil the transaction demand. At 20 warehouses the database will be I/O bound.

We were unable to obtain consistent tpmC measurements for timestamp CDC, trigger CDC and DBMS log CDC at the 20 warehouse configuration. Moreover, the database became so I/O bound that the results were unrepeatable. We were able to obtain tpmC measurements for the TAAR CDC mechanism and the control experiment. This suggests that the pull CDC mechanisms exacerbate I/O issues, in an already I/O bound database.

Figure 7.4 shows the tpmC performance for TAAR CDC mechanism and the control experiment at the 20 warehouse configuration. Again the TAAR approach slightly outperforms the control experiment. Figure 7.4 shows no relationship between the number of clients and the tpmC rate when there are 20 warehouses.

Figure 7.5 plots the running average graphs for the TAAR CDC and control experiments. The tpmC running average graphs (Figures 7.5a and 7.5b)show that the database was I/O bound even without pull CDC in operation.

7.2 Capture Latency

Capture latency is a measure of how much time there is between a change being committed to the database, and a change being captured. For each experiment scenario capture latency was measured for each change that was made to the database over a one hour period. Each experiment scenario produced a capture latency data set.

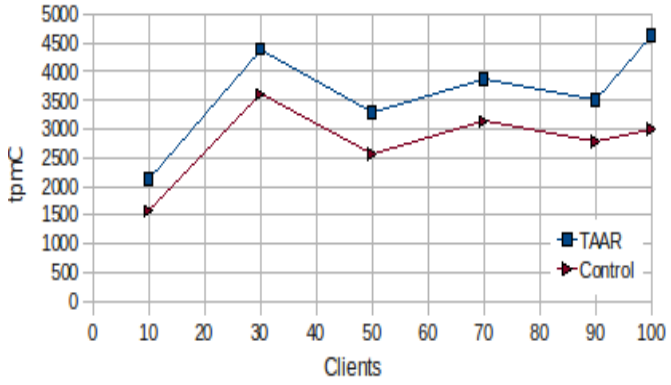
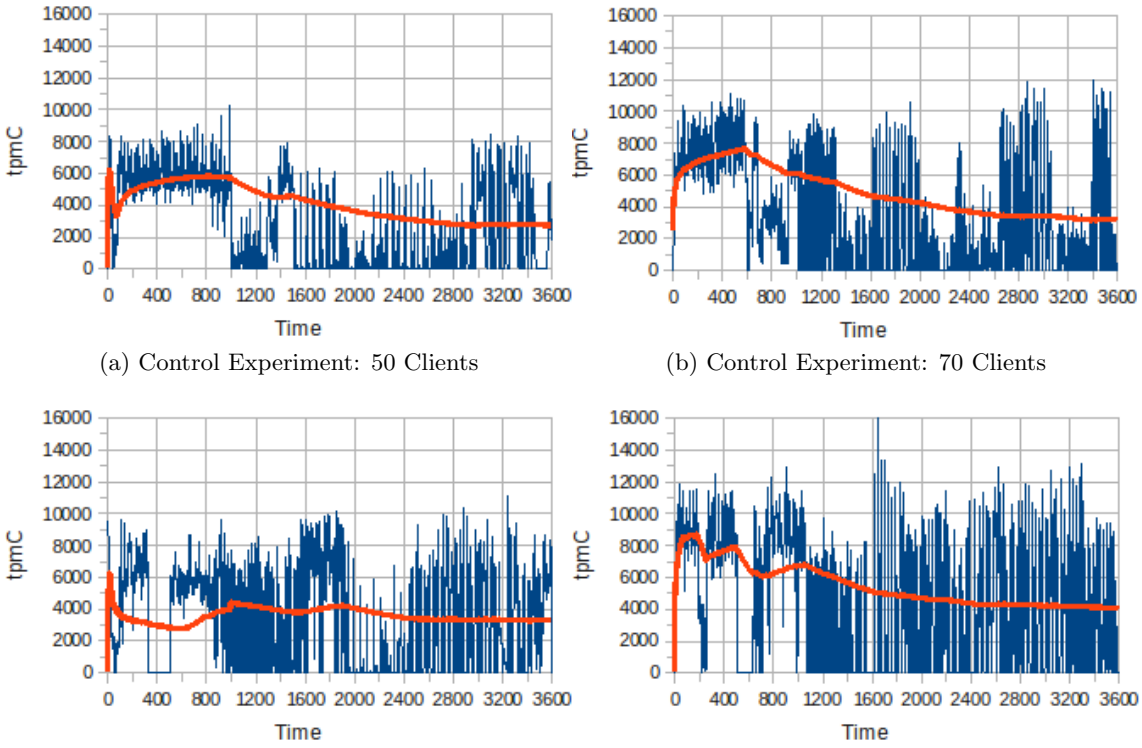


Figure 7.4: tpmC for TAAR CDC and Control Experiment for Various Client Scenarios; 20 Warehouses



(a) Control Experiment: 50 Clients

(b) Control Experiment: 70 Clients

(c) TAAR CDC: 50 Clients

(d) TAAR CDC: 70 Clients

Figure 7.5: tpmC at 2 Second Intervals; 20 Warehouses

We were unable to obtain results for the pull CDC architectures when the mechanical latency was zero. Therefore, we had to add a 50 millisecond mechanical latency between successive poll operations. Without this mechanical latency we found that the DBMS was unable to function properly for us to obtain any meaningful or repeatable results. By adding a 50ms mechanical latency, we were able obtain results to test our hypothesis. Furthermore, adding a mechanical latency to the pull CDC mechanisms meant that according to Definition 2, the pull CDC techniques are not capable of real-time CDC. However, despite the mechanical latency, we shall show in the following sections that pull CDC can

still be used to provide real-time CDC that is comparable to the TAAR architecture.

Additionally, inherent complexities in computer systems meant that capture latency is a random variable. To obtain useful information from the capture latency data sets, we performed a statistical analysis on each data set. The following sections detail the statistical analysis and produce a set of statistics to show how much capture latency each CDC mechanism causes.

7.2.1 Normal Conditions

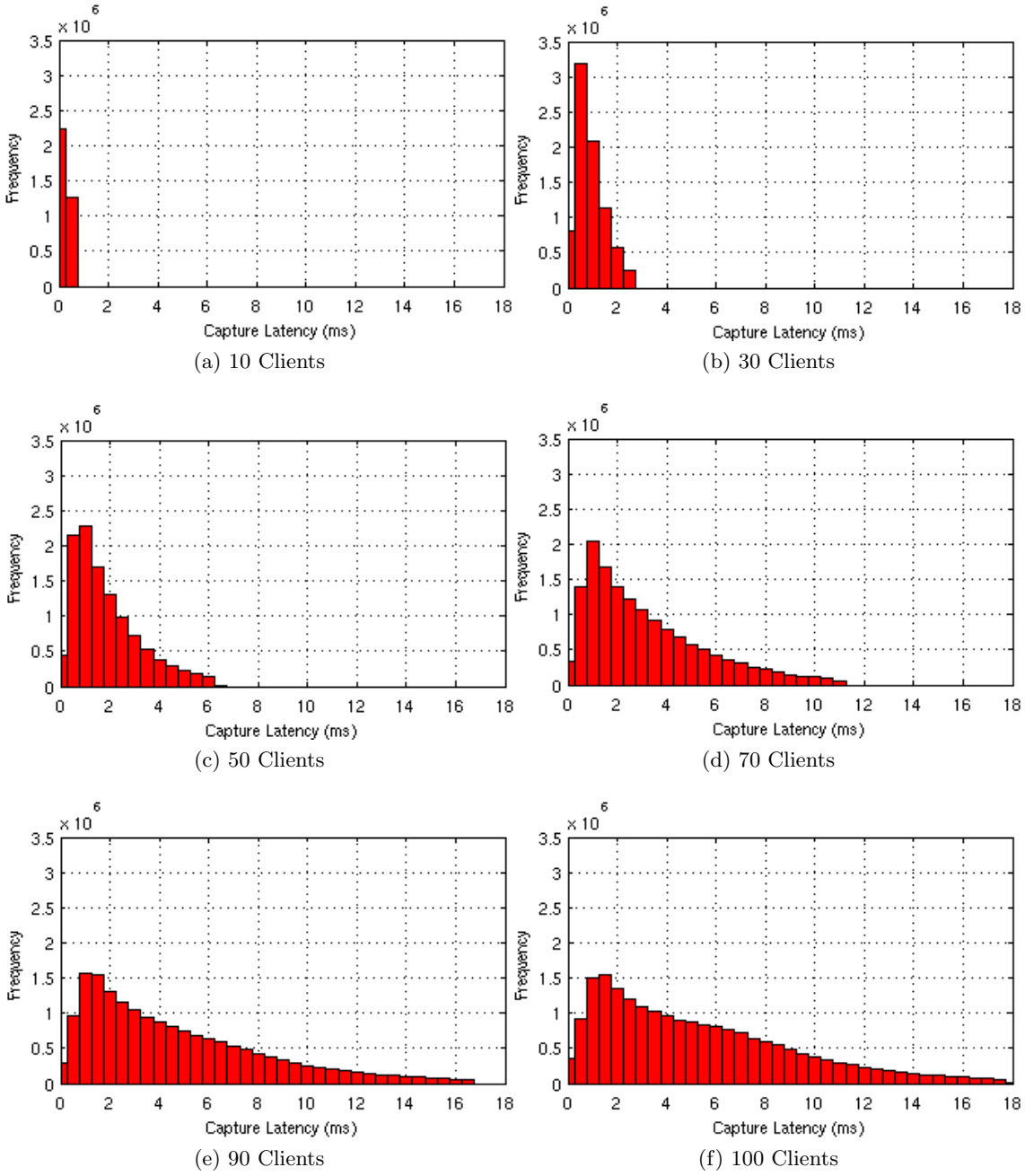
In order to do a statistical analysis on the capture latency data sets, we first had to fit the data sets to a distribution. We initially found that the data sets were heavily skewed, due to a small percentage of outliers in each data set. Fluctuations in the Java virtual machine, e.g garbage collection, will briefly consume resources, which might slightly delay the capturing of some changes. These resource spikes are likely to produce outliers in the data sets. To get a better fit for the data sets, we identified and removed outliers. We defined an outlier using the set notation in Equation 7.1. Essentially any capture latency that is three times greater than the data set's interquartile range is classified as an outlier. Table 7.2 shows the total number of data points in each capture latency data set for all of the CDC mechanisms as various client scenarios. We found that the outlier set in Equation 7.1 contained approximately 3% of the total data points for each capture latency data set. Three percent of each data set is a small number of outliers, considering each data set has over 2 million data points.

Clients	**TAAR**	**Timestamp**	**Trigger**	**DBMS Log**
10	3650584	3703325	3402091	2763283
30	8466966	7642241	4995000	5325397
50	12218250	10633030	4977758	8398511
70	15706544	7192224	3807616	5939104
90	18441696	5597241	2949899	-
100	21138362	6059823	2273584	-

Table 7.2: Total Number of Capture Latency Data Points in each Data Set

$$outlier = x \in dataset | x \geq 3 * IQR \tag{7.1}$$

Figures 7.6-7.9 show the distribution of the capture latency data sets with outliers removed. We shall now discuss each figure.



(a) 10 Clients (b) 30 Clients

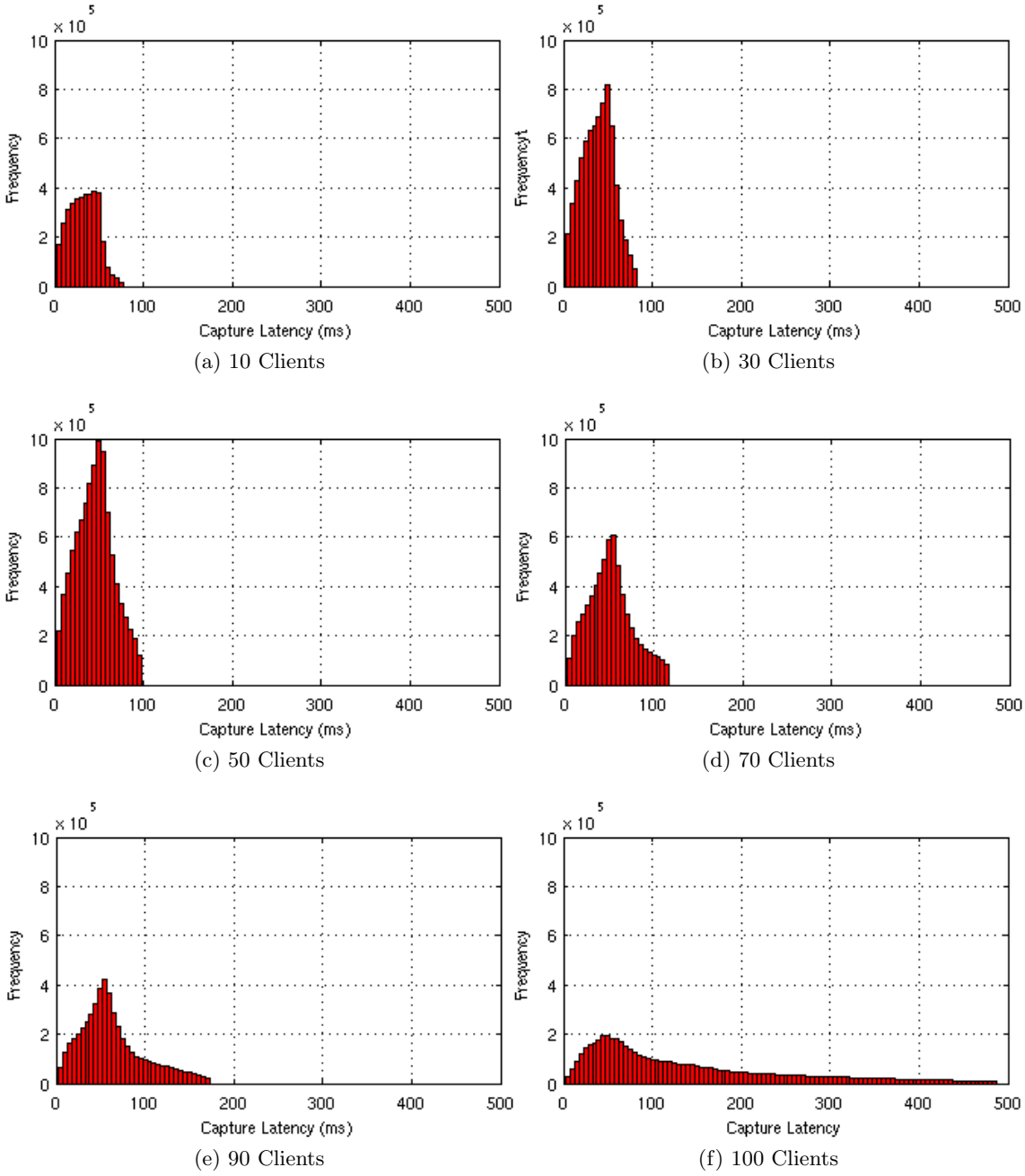
(c) 50 Clients (d) 70 Clients

(e) 90 Clients (f) 100 Clients

Figure 7.6: Capture Latency Histogram Plots for TAAR CDC; 10 Warehouses

The capture latency distributions in Figures 7.6-7.8 are positively skewed, some more than others. The distributions are skewed due to the fact that we are measuring time and the time measurement starts from zero. Due to the nature of time, it is impossible for a change to be captured in less than zero time. Therefore capture latency must always be a positive number, which accounts for the skewness in the data sets.

We found that the distributions in Figures 7.6-7.8 best fit a Weibull distribution. The Weibull distribution has two parameters: shape (β) and scale (η). The shape indicates the shape of the distributions slope. The scale indicates the height of the peak and the size



(a) 10 Clients (b) 30 Clients

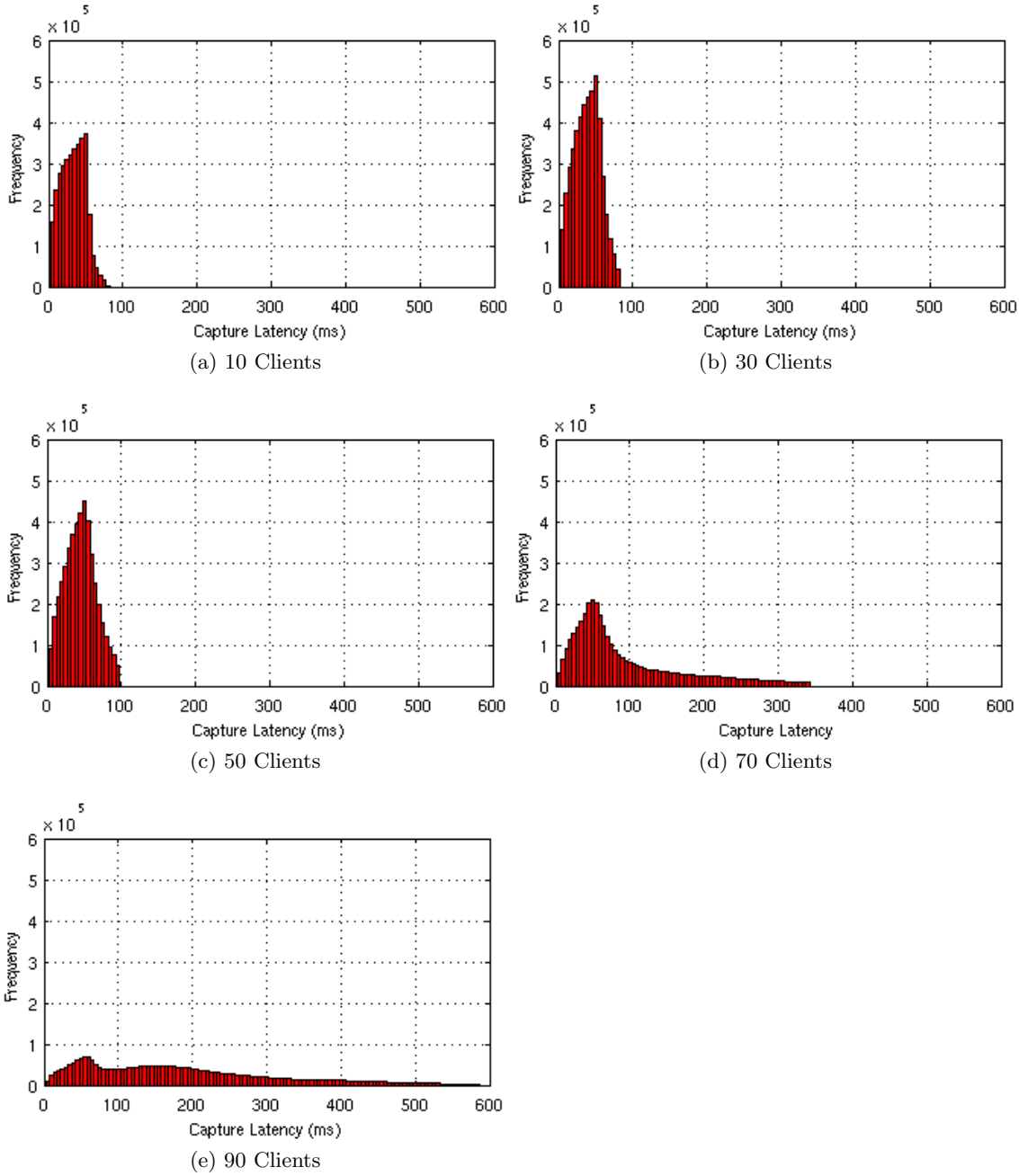
(c) 50 Clients (d) 70 Clients

(e) 90 Clients (f) 100 Clients

Figure 7.7: Capture Latency Histogram Plots for Timestamp CDC; 10 Warehouses

of the tail. Using Maximum Likelihood Estimation (MLE) we obtained the two Weibull parameters for each data set, which we shall later use in the distributions cumulative density function.

For DBMS log CDC, the distributions in Figures 7.9a- 7.9c are normally distributed. Even though capture latency must still be greater than zero in the DBMS log CDC scenario, it appears that the capture latency in DBMS log CDC is a couple of magnitudes greater than in the other scenarios, therefore, capture latency is less bound by zero. This suggests that something fundamental to DBMS log CDC causes capture latency to be so



(a) 10 Clients

(b) 30 Clients

(c) 50 Clients

(d) 70 Clients

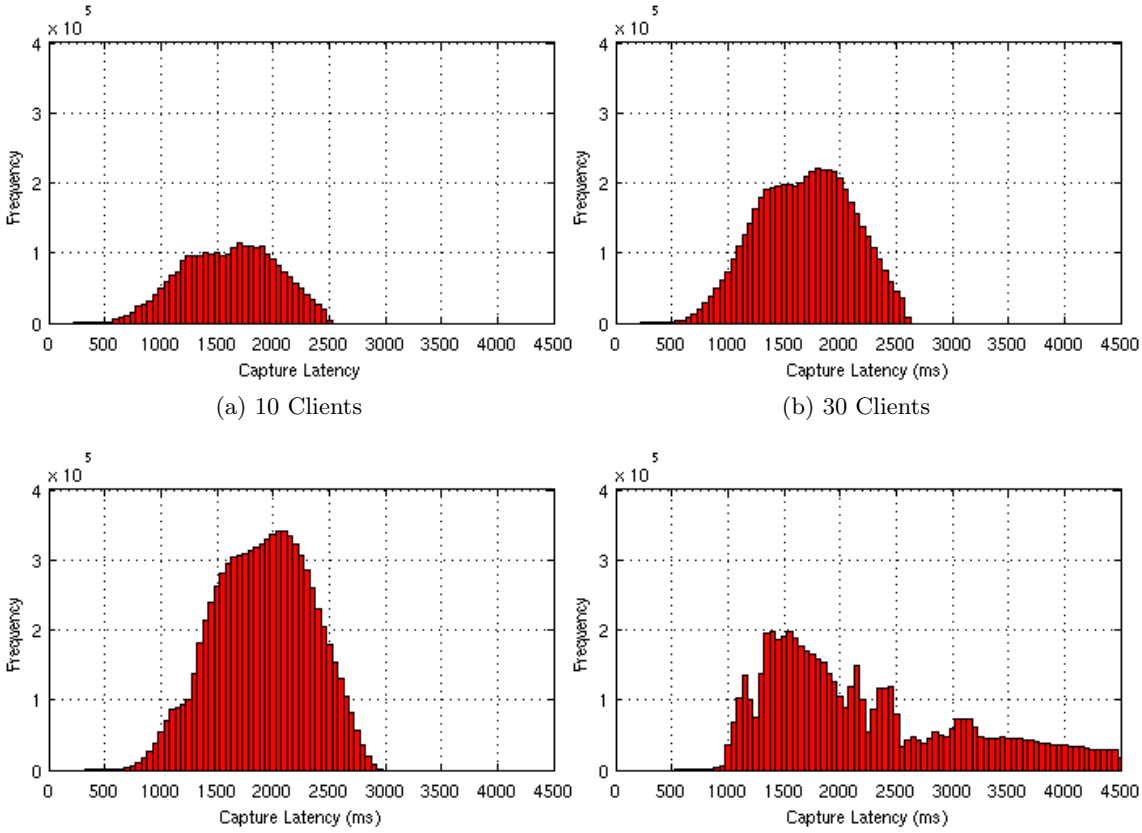
(e) 90 Clients

Figure 7.8: Capture Latency Histogram Plots for Trigger CDC; 10 Warehouses

high. The distribution in Figure 7.9d best fits a Weibull distribution, although it is not a perfect fit.

To gain further insight into capture latency we obtained the interquartile range for each capture latency data set. Table 7.3 shows the interquartile range for the six capture latency data sets for all of the experiment scenarios.

Considering the interquartile range for the TAAR CDC data sets, it appears that as the number of clients increases, capture latency increases. Moreover there appears to be a relationship between the number of clients and the capture latency ($f(x) = 0.01x^{1.44}$)



(a) 10 Clients

(b) 30 Clients

(c) 50 Clients

(d) 70 Clients

Figure 7.9: Capture Latency Histogram Plots for DBMS Log CDC; 10 Warehouses

Clients	TAAR	Timestamp	Trigger	DBMS Log
10	0.2	25	26	623
30	0.8	27	27	627
50	1.8	29	29	632
70	3.3	33	88	663
90	5.0	43	180	-
100	5.7	146	192	-

Table 7.3: Interquartile Range for Capture Latency Data Sets; 10 Warehouses

Considering the interquartile range for the 6 timestamp CDC data sets, it appears that capture latency became more variable as the number of clients increased. Capture latency appears to increase linearly with the number of clients for the 10, 30, 50 and 70 client scenarios. The 90 and 100 client scenarios do not follow the linear pattern, which suggests higher numbers of clients put a significant strain on capture latency.

Considering the interquartile range for the 6 trigger CDC data sets, it appears that capture latency exhibits a similar pattern to the timestamp CDC data sets. The interquartile range for the trigger data sets show that the capture latency increased linearly for the 10, 30 and 50 client scenarios, which is similar to the timestamp capture latency data

sets. However, it appears that triggers have a different effect on the capture latency from 70 clients onward. We noted that the linear pattern was broken at the 90 client scenario in the timestamp capture latency data sets. The pattern is broken sooner in the trigger experiments at the 70 client scenario.

The interquartile range values for the DBMS log CDC data sets (see Table 7.3) show that there is a linear relationship between capture latency and the number of clients for the 10, 30 and 50 client scenarios. The 70 client scenario does not follow the same linear relationship. Apparent from the DBMS log CDC data sets is the additional overhead on capture latency compared to the other CDC mechanisms.

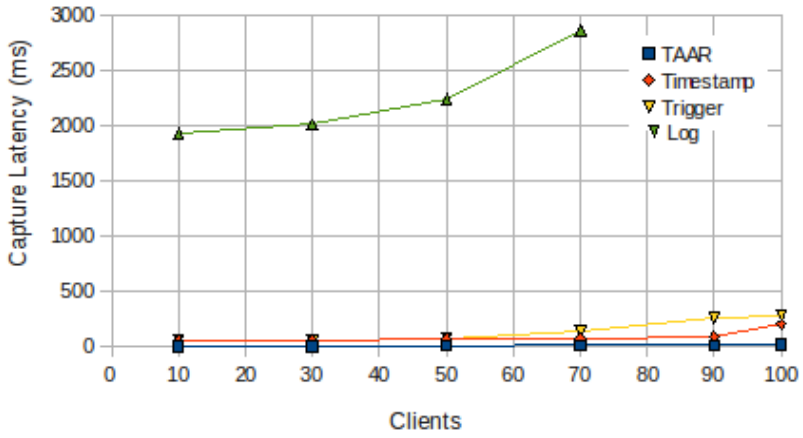
The Weibull distributions are positively skewed, so the mean capture latency will be a value closer to the tail of the distribution than the main bulk of the data. Instead of using the mean capture latency to represent the data set's capture latency; we used each distribution's cumulative distribution function, to deduce the time by which there is a 75% chance that a change has been captured, for each CDC mechanism. Figure 7.10 shows the time by which there is a 75% chance that a change has been captured. Figure 7.10a further demonstrates the difference in capture latency between the four CDC mechanisms, with DBMS log CDC having significantly more capture latency. Figure 7.10b shows the similarities in capture latency between timestamp CDC and trigger CDC for client scenarios of 50 and less, it also highlights the difference in capture latency to TAAR CDC. Figure 7.10c further demonstrates the relationship between the number of clients and capture latency in the TAAR CDC experiments.

An observation from the data presented in Figure 7.10 is that TAAR CDC has the least amount of capture latency. The likely reason that TAAR CDC has less capture latency is because we had to introduce a mechanical latency of 50ms to the pull CDC techniques, in order to obtain the results. Without this latency, we were unable to get repeatable results when using the pull CDC mechanisms.

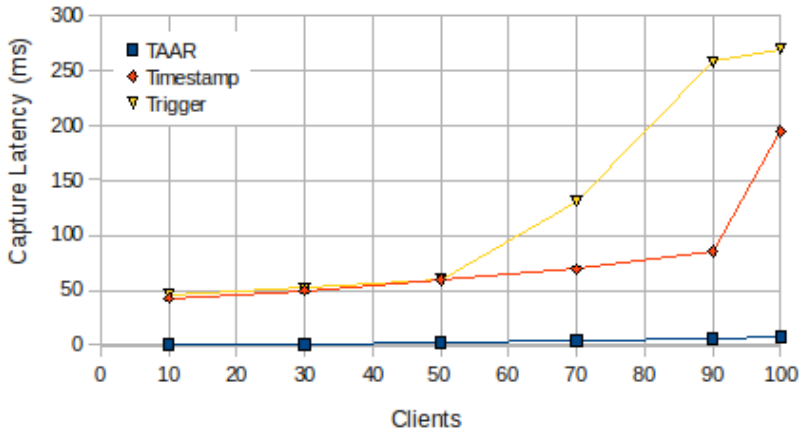
Adding a mechanical latency to the pull CDC mechanisms meant that according to Definition 2, the pull CDC techniques are not capable of real-time CDC. However, despite the mechanical latency, the results in Figure 7.10 suggest that timestamp and trigger based pull CDC architectures are comparable to capture latency in TAAR CDC, especially when the number of simultaneous clients is 70 or less. Capture latency in DBMS log CDC on the other hand is not comparable to capture latency in TAAR CDC.

To further consider that pull CDC[1] is comparable to TAAR CDC, Table 7.4 shows how much additional latency is in each of the pull CDC mechanisms when compared to TAAR CDC. The data in Table 7.4 is calculated by using the TAAR capture latency as a baseline; the capture latency for TAAR CDC is subtracted from the capture latency for each of the pull CDC techniques, which gives the latency delta between TAAR CDC and the pull CDC techniques. When the number of clients is 70 or less, the timestamp and trigger CDC approaches only add a small amount of additional capture latency. To qualify that statement, let us compare the difference between the pull CDC techniques

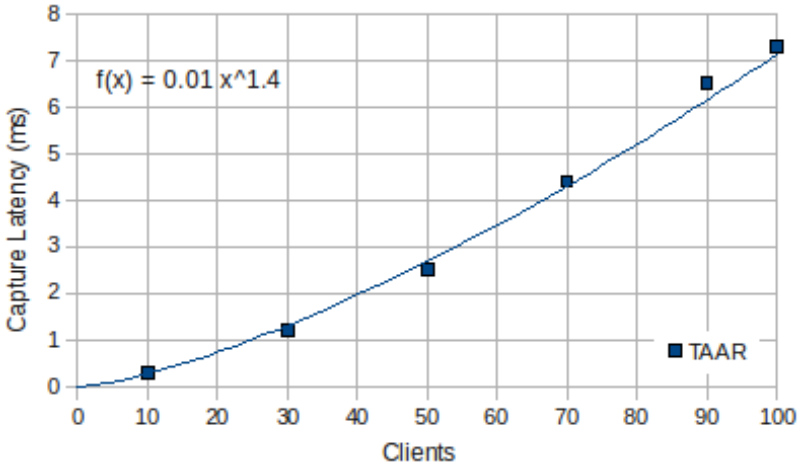
[1] when using timestamps and triggers



(a) All CDC Mecahnisms



(b) No DBMS Log CDC



(c) TAAR CDC Only

Figure 7.10: Time By Which there is a 75% Chance that a Change Has Been Captured

and TAAR, by considering the maximum possible number of transaction commits that can occur, before there is a 75% chance that a change has been captured. In other words, how much work can the database do before there is a 75% chance that a change been captured?

Clients	TAAR Baseline (ms)	Timestamp CDC (ms)	Trigger CDC (ms)
10	0.3	42.7	45.7
30	1.2	48.8	50.8
50	2.5	57.5	56.5
70	4.4	64.6	126.6
90	6.5	78.5	250.5
100	7.3	186.7	261.7

Table 7.4: Capture Latency Delta for each of the Pull CDC Techniques Where TAAR CDC is the Baseline

Clients	TAAR	Timestamp CDC	Trigger CDC	DBMS Log CDC
10	0	3	3	94
30	0	7	5	156
50	1	12	5	306
70	1	10	10	255
90	3	10	18	-
100	3	24	19	-

Table 7.5: Number of Possible Transaction Commits in Capture Latency Delta

To work out how many transactions were committed in the time by which there is 75% chance that a change has been captured, we first calculated the total number of transactions per millisecond. Table 7.1 presents the total number of transactions that were committed per second for each of the experiment scenarios. Given the data in Table 7.1, we can calculate for each scenario how many transactions were committed every millisecond, by dividing each value by 1000. Using the average number of transactions per millisecond data, we could then produce Table 7.5, which shows how many transactions are likely to be committed in the time by which there is 75% chance that a change has been captured.

When the difference in capture latency is considered in terms of how much transactional work the system can do in the given time frame, it becomes apparent that the mechanical latency in the pull CDC architecture when using timestamps or triggers is insignificant at 70 clients or less. Therefore, we argue that at the lower client levels the difference in capture latency is relatively minimal. Moreover, despite the adding of a mechanical latency the capture latency between pull and push CDC architectures is comparable. However, it is clear that pull CDC when using DBMS log CDC is not comparable to TAAR CDC.

7.2.2 Extreme Conditions

This section presents the results for capture latency when the database is under extreme conditions (1 warehouse and 20 warehouses). For the pull CDC mechanisms timestamp CDC, trigger CDC and DBMS log CDC, we were unable to obtain repeatable measurements for the capture latency when the TPC-C experiments were set up with 20 warehouses. We do not include timestamp CDC, trigger CDC and DBMS log CDC 20 warehouse results in this section. We discuss why we were unable to obtain these results in

Chapter 8. We were able to obtain capture latency results for TAAR CDC at 20 warehouses. We were able to obtain latency results for all four CDC mechanisms when the TPC-C experiments were set up with 1 warehouse.

To gain useful statistics from the extreme condition capture latency data sets, we followed the same procedure we took for the previous capture latency data sets. First we removed outliers, using the outlier definition in Equation 7.1. We then used Matlab to find a fit for each of the data sets. We found that the data sets either fit a normal distribution, or a Weibull distribution.

7.2.2.1 1 Warehouse Configuration

In the one warehouse scenario we were able to obtain consistent capture latency data sets, for all of the CDC mechanisms at the 6 different client scenarios. Table 7.6 shows the interquartile range of the 6 data sets, for each CDC technique when the TPC-C experiments were set up with one warehouse. Unlike in the 10 warehouse scenario, there is not a significant relationship between the number of clients and capture latency. The capture latency appears to level off in these data sets. Again, the capture latency in the DBMS log CDC data sets is at least two orders of magnitude greater than the capture latency in the other CDC approaches.

Clients	TAAR	Timestamp	Trigger	DBMS Log
10	0.227	22	22	357
30	0.404	24	24	363
50	0.451	24	24	368
70	0.453	24	25	375
90	0.475	24	25	378
100	0.526	24	26	381

Table 7.6: Interquartile Range for Capture Latency Data Sets; 1 Warehouse

7.2.2.2 20 Warehouse Scenario

The capture latency data sets from the TAAR CDC 20 warehouse scenario fitted a Weibull distribution. Table 7.7 shows the interquartile range for these data sets. The range of capture latency is similar to what we observed in the 10 warehouse configuration for the TAAR scenario. Again there is a relationship between the number of clients and the capture latency ($f(x) = 1.01x^{1.44}$), which suggests an I/O bound database does not affect capture latency in the TAAR CDC mechanism.

Figure 7.11 shows the correlation between number of clients and the time by which there is a 75% chance that a change has been captured. Again these results exhibit a similar relationship between the client levels and capture latency, which we saw for the 10 warehouse configuration.

Clients	TAAR
10	0.224
30	0.933
50	2.132
70	3.967
90	5.700
100	6.405

Table 7.7: Interquartile Range for TAAR Capture Latency Data Set; 20 Warehouses

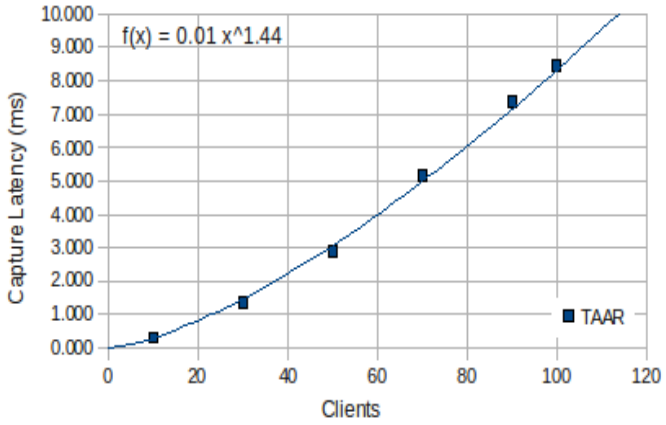


Figure 7.11: Time By Which There is A 75% Chance that a Change Has Been Captured TAAR CDC; 20 Warehouses

7.3 CPU Usage

The CPU usage results show the average amount of CPU resources that were consumed during each experiment scenario, for each of the CDC architectures.

7.3.1 Normal Conditions

Figure 7.13 shows the CPU usage results for the all of the CDC scenarios. For the lower client scenarios all of the CDC techniques use approximately the same amount of CPU resources. As the number of clients increases the TAAR CDC mechanism exhibits more CPU usage than the other CDC mechanisms. The results show that the pull CDC mechanisms do not use excessively more CPU than the control experiments.

7.3.1.1 TAAR CPU Usage

Figure 7.13 shows a comparison of TAAR CPU usage when CDC is turned on versus TAAR CPU usage when CDC is turned off, for additional comparison the CPU usage results for the control experiments are included. It is evident that the TAAR approach uses more CPU than the control experiments. Furthermore, the TAAR uses more CPU resources when CDC is turned on, which indicates there the CDC logic requires extra processing.

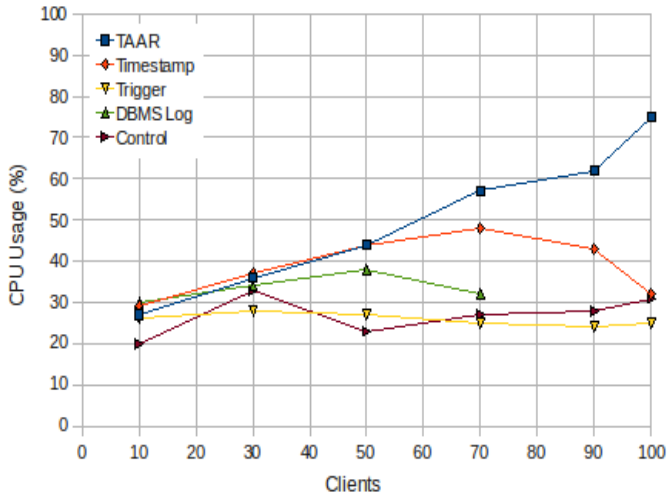


Figure 7.12: CPU Usage for each CDC Mechanism and Control Experiment for Various Client Scenarios; 10 Warehouses

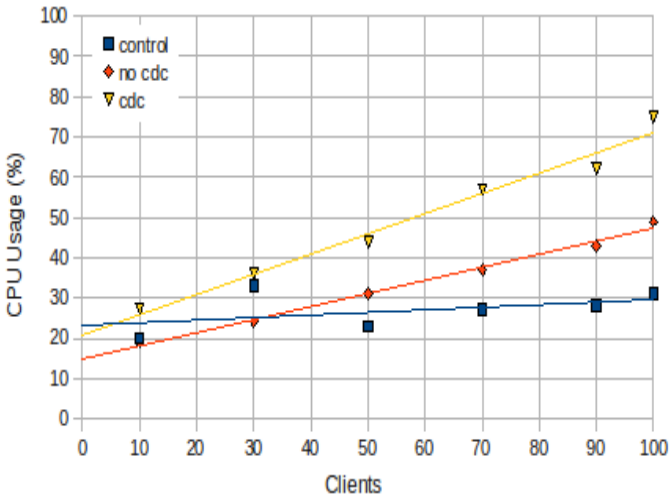


Figure 7.13: CPU Usage for TAAR with CDC and TAAR without CDC and Control Experiment

7.3.2 Extreme Conditions

7.3.2.1 1 Warehouse Configuration

Figure 7.14 shows the CPU usage results all of the CDC mechanisms and the control experiment for the 1 warehouse configuration. At the 1 warehouse configuration the pull CDC architectures use slightly more CPU usage than TAAR based CDC. This is the opposite of what we observed for the 10 warehouse scenario, and suggests that a database that is bound by contention requires more CPU resources. Also observable from these results is that the TAAR uses less CPU resources when the database is bound by contention.

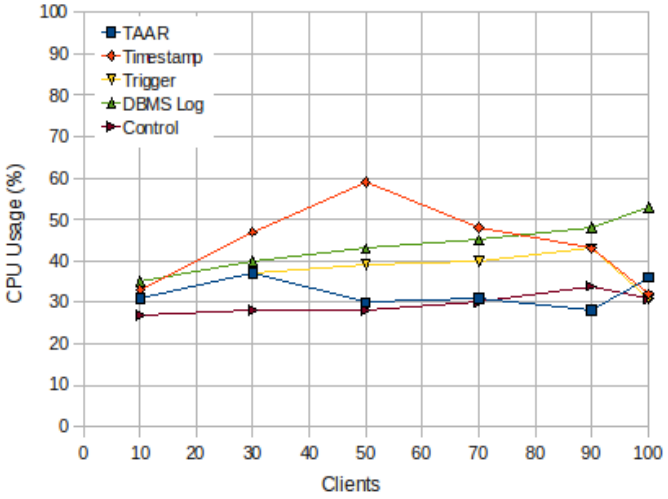


Figure 7.14: CPU Usage for each CDC Mechanism and Control Experiment for Various Client Scenarios; 1 Warehouse

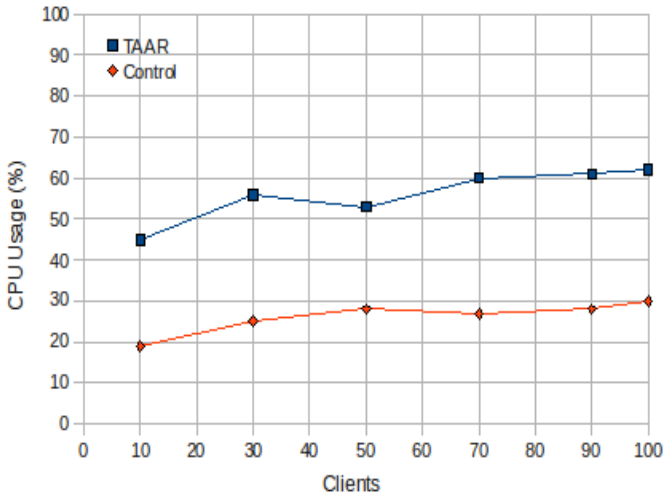


Figure 7.15: CPU Usage for each CDC Mechanism and Control Experiment for Various Client Scenarios; 20 Warehouses

7.3.2.2 20 Warehouse Configuration

Figure 7.15 shows the CPU usage results for TAAR and the control experiment for the 20 warehouse configuration. The control experiments exhibits CPU usage that is similar to what we observed in the 10 warehouse configuration, which suggests an I/O bound database uses no additional CPU usage. The TAAR experiments exhibit high CPU usage that is similar to what we observed in the 10 warehouse configuration, which possibly suggests that the TAAR are CPU bound, regardless of the database being I/O bound.

7.4 Summary

This Chapter presented the results that were obtained from our CDC evaluation experiments. Results were presented for three variables: 1) transaction rate; 2) capture latency; 3) CPU usage.

The transaction rate results, under normal conditions (10 warehouses), demonstrated that when the client load was between 10 and 30 clients, the CDC mechanisms had a nominal impact on the transaction rate, compared to the control experiment. As the number of concurrent clients increased up towards 50 and beyond, it became apparent that pull CDC mechanisms impacted the database's transaction rate, whereas the TAAR push CDC mechanism continued to demonstrate a nominal overhead on the transaction rate. At the higher client levels, between 70 and 100, the benchmarks with pull CDC mechanisms, committed four times fewer transactions per second than the control benchmark and the TAAR benchmark. Furthermore, we were unable to obtain consistent results for the DBMS log CDC benchmarks at 90 and 100 clients. Coupled with the data in the tpmC performance graphs, the transaction rate results suggest that at higher client levels, the database is overwhelmed with I/O requests from both the transactional workload and the pull CDC workload. The results for the 1 and 20 warehouse configurations further support this observation. At the 1 warehouse configuration, where there is not enough currency in the system to allow the database to be overwhelmed by transactional I/O requests, we observed that both push and pull CDC mechanisms were comparative to each other in terms of the transaction rate. However, at the 20 warehouse configuration, which allows plenty of concurrency, thus creating a lot of transactional I/O, we observed that the additional I/O from the pull CDC mechanisms caused the database to become unstable and we were unable to obtain repeatable results. On the other hand, the results for TAAR CDC demonstrate that a push CDC mechanism does not place an additional workload on the database, thus it has a transaction rate that is comparable with the control experiment, even outperforming the control experiment on occasions.

For the capture latency experiments, we had to introduce a 50 millisecond mechanical latency into the poll operation for the pull CDC mechanisms. We found that we were unable to obtain repeatable results when there was no mechanical latency between poll operations. This naturally meant that the capture latency was longer in the pull CDC experiments. Moreover, the TAAR CDC mechanism had no mechanical latency and only had processing latency, so its overall capture latency was less. Despite the mechanical latency, the capture latency results for the pull CDC mechanisms of triggers and timestamps are comparable to the capture latency in TAAR CDC. The results become comparable when one considers how much transactional work can be done in the additional time the pull CDC mechanisms take to capture the changes. The amount of transactional work that can be done in the time difference is nominal for the lower client loads. However, when the client load is 70 or above, the number of transactions that can be done in the time difference increases by roughly an order of magnitude. DBMS log CDC on the other

hand, had significantly more capture latency and was not comparable to TAAR CDC, or even to the other pull CDC mechanisms.

The CPU usage results showed that for up to 30 clients, the CDC mechanisms all consumed similar amounts of CPU resources. From 50 clients onwards, the pull CDC mechanisms demonstrated less CPU consumption than the TAAR CDC mechanism. An observation from the transaction rate results is that from around 50 clients onwards, the pull CDC mechanisms become I/O bound. When the database is I/O bound, there is less work for the CPU to do, as processes are predominantly waiting on disk operations to complete. The CPU results for the pull CDC mechanisms tie in with the transaction rate results, by demonstrating the aforementioned behaviour. However, the results for the TAAR mechanism clearly show that it consumes the most CPU resources. An interesting observation on the TAAR CPU usage is the difference in CPU consumption between TAAR with CDC enabled, and TAAR with CDC disabled. When CDC is enabled, the TAAR uses more CPU resources.

The next chapter examines these results further and applies theoretical knowledge to the observations, in order to give reasons for why observed the results that we did. Additionally, the next chapter considers the implications of the results.

Chapter 8

The Impact of Implementing Real-Time Change Data Capture

This chapter discusses the results that were presented in Chapter 7. The results are discussed in the context of real-time data warehouse systems, and the impact each benchmarked CDC mechanism may have on a real-time data warehouse system. The impact a CDC mechanism can have on a current system is dependent upon the CDC mechanism used. To recap, Chapter 2 put forward a theory that pull CDC mechanisms must poll the database of interest at a high frequency in order to capture changes in real-time. Subsequently, high frequency polling will have an adverse impact on the database's transaction rate, which is undesirable. Therefore, when concerning pull CDC mechanisms, the following discussion uses the results to weigh up the impact high frequency polling has on the database's transaction rate. Push CDC mechanisms on the other hand do not poll a database resource in order to obtain change data. Instead, push CDC mechanisms capture data changes when they are on route the database. Recapping on Chapter 2, the reason push CDC mechanisms are seldom implemented is because they are intrusive towards existing systems, thus the impact of push CDC is the effort required to modify existing systems, in order to accommodate the CDC mechanism. Chapter 3 presented a new push CDC mechanism that can minimise the intrusion, and section 3.1.3 addresses the issue of intrusion, and how TAAR CDC can minimise it. However, it is possible that more work and effort is required to implement TAAR CDC, than is required to implement a pull CDC mechanism. Therefore, when concerning push CDC mechanisms, the following discussion uses the results to weigh up the impact of the effort required to implement a push CDC mechanism. This is done by comparing the performance data for the TAAR CDC approach, relative to the pull CDC mechanisms; the TAAR CDC approach is viable when its performance gains out way the potential efforts required to implement the CDC mechanism.

An observable trend in the results is that capture latency is correlated to the number of clients that are running in a given experiment scenario. The number of simultaneous clients is essentially a measure of concurrency. The discussion on the results will focus

around concurrency, and separate the results into low concurrency (10-30 clients), medium concurrency (50-70 clients) and high concurrency (90-100 clients).

This chapter is mapped out as follows, Sections 8.1, 8.2 and 8.3 discuss in greater detail the impact of real-time CDC at low, medium and high levels of concurrency respectively. Section 8.4 discusses why capture latency was significantly higher in DBMS log CDC. Section 8.5 links the pull CDC results back to work that has previously been conducted in the field. Section 8.6 provides a summary of the concurrency discussion.

8.1 Impact of Real-Time CDC at Low Concurrency Levels

At low levels of concurrency, pull CDC architectures when using triggers or timestamps, have a capture latency that is comparable to push CDC, even though the pull architecture has a 50ms mechanical latency. In this section we use the low concurrency results to determine the impact of high frequency polling in terms of transactional throughput and CPU usage. The results are contrasted to the impact of implementing push CDC.

The results in Chapter 7 (Figure 7.1) show that the pull CDC architectures have a minor impact on the transaction rate when concurrency is low, especially when the pull architecture is using the timestamp approach. Based on the results (Figure 7.1) Table 8.1 presents the overhead each CDC technique places on the transaction rate when compared to the control experiment. Furthermore, based on the CPU results at 10 warehouses (Figure 7.12), Table 8.2, shows how many more times each CDC technique used the CPU resources when compared to the control experiment.

The TAAR CDC mechanism at low concurrency placed no overhead on the transaction rate (Table 8.1). In fact the results show that it is more efficient to execute a transaction through the TAAR than by going directly to the database. The theory for why that is, has previously been discussed in chapter 5. The fact that the TAAR are on the same server as the DBMS reduces network communication overheads between the DBMS and the transaction logic. The CPU overhead results (Table 8.2) suggest that at low concurrency levels the TAAR does not use significantly more CPU resources than the pull CDC techniques. At low levels of concurrency the results show that the TAAR is capable of real-time CDC, without placing significant overheads on the system's resources. The

Clients	TAAR	Timestamp CDC	Trigger CDC	DBMS Log CDC
10	0.97	1.12	1.26	1.53
30	0.94	1.16	1.85	2.16

Table 8.1: tpmC Overhead for each of the CDC Techniques

Clients	TAAR	Timestamp CDC	Trigger CDC	DBMS Log CDC
10	1.35	1.45	1.3	1.5
30	1.09	1.12	0.85	1.03

Table 8.2: Additional Amount of CPU each CDC Technique Uses

impact of TAAR here is that of implementing the services if one does not have such an infrastructure in place.

When the timestamp CDC technique is used the transaction rate drops by 1.12 times at the 10 client scenario, and 1.16 times at the 30 client scenario. When the trigger CDC technique is used the transaction rate drops by 1.26 times at the 10 client scenario, and 1.85 times at the 30 client scenario. At these low concurrency scenarios it appears that a high polling frequency does not have a significant impact on the transaction rate. Furthermore, the results in Table 8.1 suggest that the two pull CDC techniques do not place significant overheads on the CPU resources.

At low levels of concurrency the results do not affirm our hypothesis. Except for DBMS log CDC, which we discuss in Section 8.4, the pull CDC techniques and high frequent polling do not have a significant impact on the transaction rate, and have a capture latency that is comparable to push CDC. Moreover, at low concurrency levels one can use pull CDC to give comparable capture performance to push CDC and with less of an impact to existing systems. For example, consider the impact of building a real-time CDC architecture from scratch, for a system where low concurrency is anticipated. The timestamp and trigger pull architectures are likely to be more viable than TAAR CDC, because it is likely to cost more time to build a TAAR CDC system from scratch than to implement triggers or add timestamps to the schema and then build a capture tool to capture the data. Finally, even though the pull CDC techniques impact the transaction throughput, the impact is small enough, that one might be able to tolerate it for the sake of quick and simple real-time CDC. However, if one already has the TAAR or a set of DAS in place then the CDC logic can potentially be added for minimal impact and maximum gain.

8.2 Impact of Real-Time CDC at Medium Concurrency Levels

At medium levels of concurrency, the pull CDC architectures based on triggers or timestamps, continue to demonstrate capture latency times that are comparable to push CDC. However, at the medium level of concurrency the tpmC results (Figure 7.1) suggest that the pull CDC techniques begin to have a significant overhead on the transaction throughput, and low mechanical latency (50ms) starts to become unsustainable.

For the purpose of the discussion on medium concurrency Table 8.3 presents the overhead each of the two pull CDC techniques, and TAAR CDC have on the transaction rate at medium levels of concurrency. Table 8.4 presents how many more times each CDC technique used the CPU resources when compared to the control experiment at medium levels of concurrency.

At medium levels of concurrency the TAAR CDC architecture remains a viable approach for real-time CDC. The TAAR places no overhead on the transaction rate (Table 8.3). However, the CPU usage results (Table 8.4) suggest that the TAAR are starting

to use more CPU resources than the other CDC approaches. We discuss the implications of additional CPU usage in Section 8.3.

When the timestamp CDC technique is used the transaction rate drops by 1.91 times at the 50 client scenario, at this level of medium concurrency, timestamp CDC remains a viable approach for doing real-time CDC. The impact pull CDC has on the transaction rate at this level, may still outweigh the impact of implementing TAAR from scratch.

The remaining scenarios at the level of medium concurrency start to place a significant overhead on the transaction rate. An observation from the results (Figure 7.1) is that tpmC values for the pull CDC techniques begin to significantly drop away from the control experiment at about 50 clients onwards, which is the medium level of concurrency.

The tmpC running average graphs (Figure 7.2) provide evidence to the theory that DBMS cannot handle the I/O for both a high transaction rate and high frequent polling. Consider the 70 client scenario. The control experiment (Figure 7.2b) demonstrates a few spikes of low performance, which we have previously identified to be caused by redo log switches, otherwise the tpmC performance is consistent. There is an observable difference between the tpmC running average graphs for the control experiment and the experiments where pull CDC is in operation.

The tmpC measurement for timestamp CDC becomes inconsistent from approximately 1400 seconds onwards (Figure 7.2f). The tmpC measurement for trigger CDC becomes becomes inconsistent from approximately 800 seconds onwards (Figure7.2h). The reason tpmC becomes inconsistent for the pull CDC scenarios, is because the DBMS becomes I/O bound and cannot write transactions to disk fast enough. Moreover, the data suggests that the DBMS's buffers are not being flushed to disk fast enough.

A database buffer is a memory store for change data that needs to be written to disk. Writers are responsible for clearing the buffers by writing changed data blocks to disk. If the writers cannot keep up with demand for new changes, then the buffers will become full, which slows down subsequent DML operations [93]. At medium levels of concurrency and beyond, the higher demand for data change operations will mean the buffers fill up quicker. The CDC polling operation will be frequently placing read locks on the data every 50 ms, which conflicts with writers who want to place a write lock on the data [45]. If a writer has to wait for a write lock, then the DBMS write buffers will fill up more quickly,

Clients	**TAAR**	**Timestamp CDC**	**Trigger CDC**	**DBMS Log CDC**
50	0.94	1.14	2.87	1.77
70	0.94	2.09	3.96	3.51

Table 8.3: tpmC Overhead for each of the CDC Techniques

Clients	**TAAR**	**Timestamp CDC**	**Trigger CDC**	**DBMS Log CDC**
50	1.91	1.91	1.17	1.65
70	2.11	1.78	0.93	1.19

Table 8.4: CPU Overhead for each of the CDC Techniques

which ultimately slows down the transaction rate. At the medium to high concurrency levels, a lot of change data is being generated. At the start of the experiments the buffers will not be full and the DBMS will be able to maintain a high transaction rate. However, the buffers soon become full, which is why we see the tpmC fluctuate so significantly in the running average graphs for pull CDC. This is not the case at the lower concurrency levels, because there are not enough data write requests being generated to fill the buffer with change data, so there is less impact on the transaction rate if the poll requests block the writers.

A further example of how quickly the buffers are filled up can be seen in the difference between the trigger CDC at 50 clients (Figure7.2g) and at trigger CDC at 70 clients (Figure7.2h). One can see that the DBMS becomes I/O bound sooner for the 70 client scenario (800 seconds) than the 50 client scenario (1300 seconds). The reason the 70 client scenario becomes I/O bound sooner is because 70 clients will generate more transactions than 50 clients. More transactions, means more change data blocks in the buffer, and less space for data changes to stack up if the writers get blocked by CDC poll requests.

The results at the 20 warehouse database scenario also show why pull CDC is not suitable for real-time data warehousing. At 20 warehouses, the tpmC running average graphs for the control experiment (Figures 7.5a and 7.5b) shows that the database is I/O bound, without any CDC. When the database is already I/O bound, high frequent polling only exacerbates the situation. We were unable to obtain results for any of the pull CDC techniques at the 20 warehouse scenario, because the additional I/O of CDC destabilised the database. Therefore, we were unable to obtain meaningful results that were consistent, or would add any further weight to the argument.

The results at medium concurrency levels, confirm our hypothesis for why pull CDC is not suitable for a real-time data warehouse system. In order to compete with a push CDC mechanism, the polling interval has to be in the tens of milliseconds. The results show that a polling interval so low has a consequence on the database's transaction rate. At the point of medium concurrency the impact of implementing the TAAR becomes tolerable if one requires real-time CDC.

8.3 Impact of Real-Time CDC at High Concurrency Levels

At medium levels of concurrency the results demonstrated that high frequency polling has a negative impact on the database's transaction rate, but capture latency is still comparable to push CDC. At the higher levels of concurrency, the tpmC results for pull CDC (Figure 7.1) continue to exhibit the affect of high frequency polling on transaction rate. At the high level of concurrency the only feasible approach for real-time CDC is TAAR CDC.

The TAAR CDC mechanism continues to demonstrate no impact on the transaction rate at the high levels of concurrency. For the pull CDC mechanisms to have a similar transaction rate, one would have to introduce a greater amount of mechanical latency,

which would mean that the capture latency in pull CDC will not be comparable to push CDC. The fact that TAAR CDC can provide such low capture latency is due to the fact that changes are captured from the TAAR's memory context, rather than the hard disk that the DBMS is using. Therefore, no mechanical latency is required in TAAR CDC. Instead there is only processing latency, which means capture latency and processing latency are synonymous in TAAR CDC.

An observable fact from the results (Figure 7.10c) is the relationship between concurrency and capture latency. As concurrency increases, so does capture latency. Because capture latency in TAAR CDC is processing latency, we infer from the results that it takes longer to process the change data in the TAAR mechanism, when concurrency is higher. We propose that there are two possible causes that will increase capture latency in TAAR CDC: 1) concurrency issues, where there too many competing processes in the TAAR; 2) a database bottleneck, where processes are delayed by the database.

We infer from the data that concurrency issues caused capture latency to increase as the client workload increased. Table 8.5 shows the correlation between capture latency and transaction rate (tpmC), and the correlation between capture latency and client load, for the three different warehouse scenarios. Table 8.6 shows the correlation between client load and CPU usage for the three warehouse scenarios.

To demonstrate that concurrency caused additional capture latency, let us first consider the 10 warehouse scenario. For the 10 warehouse scenario, Table 8.5 shows capture latency is highly correlated to both the transaction rate, and the client load. Furthermore, Table 8.6 shows there is a strong correlation between CPU usage and client load for the 10 warehouse scenario. These results indicate that as the work load increases there are more transactions, which means the TAAR has more client requests to handle, and more transactions to execute. The increasing workload also requires more CPU usage. The affect of an increasing workload is the TAAR has more tasks to handle, which means more context switching, and each process will have less time to do its work. Ultimately, changes will be on the change queue for longer, thus capture latency increases. We rule out database issues as the cause of increasing capture latency, because the tpmC running average graphs for TAAR in Figure 7.2 show that tpmC was at a constant for the 10 warehouse scenarios, and there were no signs of an I/O bound database, which we observed for the pull CDC approaches.

To further demonstrate that concurrency in the TAAR causes increasing capture latency; let us consider the 1 warehouse scenario. Table 8.5 shows that both the tpmC and the capture latency measurements level off from roughly 30 clients onwards, thus there is a strong correlation between capture latency and transaction rate. The transaction rate levels off because as discussed in Chapter 6 the 1 warehouse scenario will lead ACID transactions to conflict, causing high levels of lock contention in the database. Because data modification transactions only had one warehouse to target, the chances of lock contention increased. High lock contention means transactions will spend more time waiting

Clients	1 Warehouse		10 Warehouses		20 Warehouses	
	tpmC	**Capture Latency**	**tpmC**	**Capture Latency**	**tpmC**	**Capture Latency**
10	4369	0.3	2088	0.3	2141	0.3
30	5402	0.5	4836	1.2	4395	1.4
50	5491	0.6	6982	2.5	3296	2.9
70	5574	0.6	8976	4.4	3871	5.2
90	5552	0.6	10545	6.5	3506	7.4
100	5577	0.6	12090	7.3	4634	8.5
tpmC Correlation		**0.99**		**0.98**		**0.56**
Client Correlation		**0.84**		**0.99**		**0.99**

Table 8.5: Capture Latency Correlation to Transaction Rate (tpmC) and Client Load for the Three Warehouse Scenarios

Clients	1 Warehouse	10 Warehouses	20 Warehouses
10	31	27	35
30	37	36	46
50	30	44	49
70	31	57	60
90	28	62	61
100	36	75	71
Correlation	*-0.08*	*0.99*	*0.98*

Table 8.6: Correlation Between Client Load and CPU Usage for TAAR Warehouse Scenarios

for running transactions to release locks, which means the number of simultaneously running transactions is reduced. Consequently the transaction handlers in the TAAR will spend more time waiting for locks to be released. A knock on effect will be that there are also less client requests. Therefore, in the 1 warehouse scenario concurrency was reduced. Reduced concurrency leads to the low levels of capture latency shown in Table 8.5 for the 1 warehouse scenario. Moreover, with concurrency reduced, the TAAR would have had less simultaneous client requests to handle, less simultaneous transactions, and will have had more time and resources to run the threads that process the change queue, thus the noticeable reduction in capture latency.

To rule out database issues being the cause of increasing capture latency, let us consider the 20 warehouse scenario. The tpmC running average graphs in Figure 7.5, exhibit similar behaviour to the pull CDC running average graphs in Figure 7.2 where the database becomes I/O bound. The tpmC results for the 20 warehouse scenario in Table 8.5 show how the transaction throughput is restricted compared to the 10 warehouse scenario. The transaction rate in the 20 warehouse scenario is worse than in the 10 warehouse scenario, because the database becomes I/O bound. Table 8.5 shows that there is no correlation between the transaction rate and the capture latency for the 20 warehouse scenario, which suggests the I/O bound database does not cause additional capture latency. Table 8.5 shows that there is a correlation between capture latency and client load for the 20 warehouse scenario. Furthermore, the capture latency in the 20 warehouse scenario is within a millisecond of the capture latency measured in the 10 warehouse scenario, which suggests the cause of increased capture latency is the same in both scenarios. The two scenarios differ in that the transaction rate in the 20 warehouse scenario is affected by an I/O bound database. The similarity between the two scenarios is the increasing client load, and thus increased numbers of simultaneous transactions. Even though the 20 warehouse scenario exhibits a lower transaction rate than the 1 warehouse scenarios, the 20 warehouse scenarios are more likely to have simultaneous transactions, because transaction will not all be targeting the same warehouse. Therefore, it is likely that the 20 warehouse scenario has concurrency issues that are similar to those seen in the 10 warehouse scenarios, which causes capture latency to increase with the client load.

The data suggests that concurrency issues cause capture latency to increase with client

load, rather than an I/O bound database. With ample warehouses for transactions to target in the 10 and 20 warehouse cases, the three tasks the TAAR has to full fill are more likely to conflict with each other as concurrency increases. Moreover, as concurrency increases, the TAAR handles more HTTP requests, and thus does more transaction capture, which means more changes are placed on the change queue. Because the TAAR is multi-threaded, the three tasks will compete with each other for database resources. As a result there will be more context switches in the TAAR, and so whilst the other tasks are handled, changes on the change queue will have to wait longer to be processed, which is why we see an increase in the capture latency, as the number of clients increased.

The CPU usage results (Figure 7.12 and Figure 7.15) confirm the extra competition for resources inside the TAAR. At both the 10 and 20 warehouse scenarios the CPU usage is noticeably higher for the TAAR scenarios, when compared to the control experiments. An interesting observation from the CPU usage results for the TAAR scenario, is the additional CPU usage in the TAAR when CDC is turned on compared to when it is turned off (Figure 7.13). The fact that the TAAR is on the same server as the DBMS means that a server with ample CPU resources is required to run TAAR CDC. The results suggest that the CPU will be a bottleneck in the TAAR CDC approach, before the I/O becomes a bottleneck.

The CPU results (Figure 7.12) for the pull CDC mechanisms show that they do not use that much CPU in comparison to TAAR. However, we feel that the CPU results for the pull CDC mechanisms need to be treated with caution. For the pull CDC mechanisms, the CPU usage increases as concurrency increases from low to medium. Then as the database becomes I/O bound, one can see from (Figure 7.12) that the CPU usage drops off. When the DBMS is I/O bound, it is likely that we see less CPU usage in the pull CDC techniques, because operations will be waiting on disk I/O thus using less CPU.

Ultimately when the concurrency levels are high, the results show that TAAR based CDC is optimal for real-time CDC. The results for the pull CDC scenarios, affirm our hypothesis that pull CDC architectures with high frequency polling are not feasible in a real-time environment.

8.4 DBMS Log CDC Performance

The capture latency results (Figure 7.10a) show that DBMS log CDC was the worst CDC mechanism for a real-time scenario. The IQR (Table 7.3) values for the DBMS capture latency data sets at the 10 warehouse configuration show there is at least two orders of magnitude additional capture latency than in the other CDC approaches. Considering the capture mechanism was the same for all three pull CDC techniques, and the polling interval was the same, we conclude that it is the identification mechanism used in DBMS log CDC that added the additional latency.

The DBMS log technique involved reading data changes from the DBMS redo log, which is a file on the disk. A limitation of our experiments was that the redo log was on

the same disk as the operational data tables. Having the redo logs on the same disk as the operational tables will mean the CDC I/O on the redo logs will interfere with transactional I/O. The running average graphs (Figure 7.2i and Figure 7.2j) show that tpmC becomes inconsistent at an earlier point in the experiments when compared to the other pull CDC techniques. Additionally we were unable to obtain tpmC results for DBMS log CDC at 90 and 100 client scenarios, as the additional I/O meant that the results became unrepeatable.

It is highly like that our set up significantly disadvantaged DBMS log CDC. Placing the redo logs on a different hard disk to the operational tables would most likely have seen the capture latency decrease. Unfortunately we did not have the luxury of an additional hard disk for storing the redo logs. However, whilst DBMS Log CDC will potentially be improved by storing the redo logs on a separate hard drive to the hard drive where the operational data is stored, we argue that the results still show that log CDC will still have more latency than TAAR CDC.

High frequency polling would still have likely prevented DBMS log CDC from having capture latency that is comparable to push CDC at the medium and higher concurrency levels, based on the fact that the mechanism would have been similar to timestamp and trigger based pull architectures. Furthermore, the change data also has to be read from the redo logs, which are stored on a hard disk. TAAR CDC obtains change data from the system's memory. Reading change data from memory will invariably be faster than reading change data from the hard disk.

8.5 Wider Implications

This section reflects on the wider implications of our results. To begin with, we compare our pull CDC results, to the previous knowledge and results in the field. We then move on to consider the wider implications our results have for practitioners in the field.

8.5.1 Timestamp CDC

The results show that timestamp CDC caused the least amount of overhead on the transaction rate amongst the pull CDC techniques. Timestamp CDC receives a lot of criticism in the literature, especially for its performance. Authors argue that rebuilding timestamp indexes will prove costly to the transactional performance [88, 81, 63]. However, our results show that timestamp CDC is the most economical pull CDC technique towards the transaction rate. Despite the obvious flaws such as being unable to capture delete statements, timestamp CDC has good performance in a real-time environment, particularly at the lower client loads.

8.5.2 Trigger CDC

Our results show that after timestamp CDC, trigger CDC caused the least amount of processing latency amongst the pull CDC techniques. HVR Software [52] suggest that

trigger based CDC places a 3% overhead on the transactions in an OLTP environment. We observed a greater overhead than 3%. However, it is possible that most of this overhead is caused by the high frequency polling rather than the triggers themselves. HVR Software [52] claimed that capture latency was approximately a minute in their system when they used triggers, so it is likely they were polling the database less frequently, which is why they observe less transactional overhead. There is slightly more processing latency in the trigger CDC technique than in the timestamp CDC technique. The cause of this extra latency is possibly from the processing of the triggers.

8.5.3 DBMS Log CDC

Because our database became I/O bound we were unable to affirm Pareek's observation that a DBMS server exhibits high CPU usage when DBMS log CDC is enabled. The DBMS spent most of the time waiting on I/O, so the CPU remained relatively idle. Although, the results could affirm Pareek's observation that DBMS log CDC will place an overhead on the transaction rate. Pareek showed there was approximately a 33% overhead on transaction rate when they used DBMS log CDC in a TPC-C experiment. Pareek do not disclose exactly how many clients he used, however, we noted approximately a similar overhead at around 10 and 30 clients. Raitto [92] also benchmark DBMS log CDC. Raitto observed that DBMS log CDC placed an 8% overhead on the transaction rate. Raitto did not use the TPC-C benchmark, but instead used their own benchmark that simulated an OLTP environment. Raitto ran their benchmark with 250 concurrent clients, which shows DBMS log CDC can perform better than what we have observed. The capture latency that Raitto reports is similar to what we observed, that 75% of data changes are captured within 1.5 seconds. However, as discussed when we reviewed Raitto's work in Chapter 2, it is unclear as to whether they see capture as a record being captured by an external tool, or just identifying a change and moving it to a staging table. Our results demonstrated the latency in both change identification and change capture.

8.5.4 External Log CDC

Although we have not compared external log CDC in this experiment, it is possible to infer from the results that TAAR will have less latency, due to the fact that it reads change data from the TAAR's memory context. However, we cannot infer whether capture latency for an external log CDC mechanism would be comparable to TAAR capture latency. It is possible that practitioners who build external log CDC tools will be able to expand on our experiments and compare their commercial tools to our results.

8.5.5 Implications for Practitioners

Considering the wider implications of our results, it is possible that our results and interpretation of them will be beneficial to those who are implementing a real-time CDC

architecture, specifically those who are interested in doing real-time CDC, whilst maintaining a high transaction rate in systems with high levels of concurrency. We see potential for the TAAR CDC system to be used in fraud analysis architectures such as the one presented by Nguyen et al. [78]. It is possible that our TAAR based CDC solution will aptly fulfil the role of Nguyen et al.'s sense service. Nguyen et al.'s sense service is currently based on a pull CDC mechanism. Our push CDC system would feed Nguyen et al.'s interpret and respond service with operational change data at a fast rate, which could reduce the time it takes for their system to make an effective decision to potentially prevent fraud.

Furthermore, Hackathorn [48] proposed that more value can be gained from operational data, if the time between a business event occurring and a decision being made, based on the analysis of that event can be minimised. Hackathorn calls this time as the window of opportunity. It is possible that practitioners can use our results to help them determine the CDC architecture that best suits their window of opportunity and system requirements. If the window of opportunity is in the order of minutes then a DAS based CDC solution should only be used if the DAS is already in place, otherwise the cost of implementing the DAS may outweigh the cost of using one of the pull CDC mechanisms. Additionally, if one has near real-time requirements, such as Cochinwala and Panagos's [24] fraud analysis system, then again it is possible that one of the pull architectures is more suitable, and less costly to implement than the DAS.

However, the TDWI [34] recognise that a near real-time solution is often the first step towards a real-time solution. The TAAR based solution offers the optimal solution for a real-time architecture, and we propose that the TAAR solution is best equipped to handle the transition from near real-time to real-time, because it can continue to scale as an organisation grows. Therefore, one may want to use TAAR for near real-time CDC, in anticipation for future growth. Another use case where one may want to use DAS based CDC for near real-time is when one has the DAS infrastructure already in place.

A limitation for our experiments on pull CDC architectures was the hard drive that the DBMS server was using. The hard disk in the DBMS server for our experiments had a rotation speed of 7200rpm. This limitation has a wider implication on practitioners who are considering a pull CDC mechanism; one may have to use a faster hard drive when deploying real-time pull CDC in a system that has medium to high concurrency. Faster hard disk speeds of 10,000rpm or 15,000rpm could have improve the performance of pull CDC architectures by quickening the DBMS's data read and data write operations. With faster write operations, the write buffers would be less likely to become overwhelmed, and a high transactional throughput could be maintained for longer. Additionally faster read operations would mean that the CDC requests would hold read locks for less time, which also aid the data writers in keeping the write buffer from overflowing.

The results for TAAR CDC showed that eventually the CPU will become a bottleneck. Like the hard disk limitations that have implications for practitioners who are considering using pull CDC architectures, CPU limitations will have implications for practitioners who are considering using push CDC architectures with TAAR. When the CPU becomes

a bottleneck in TAAR CDC, it is likely that processing latency will significantly increase. Furthermore, in a production environment there are likely to be more processes that are competing for CPU resources, which could affect TAAR CDC.

Ultimately, when choosing a real-time CDC architecture, a practitioner will have to diligently consider the expected concurrency in the system and which hardware resources are easier to upgrade and replace once limitations are reached.

8.6 Summary

The hypothesis that this thesis tests is that pull CDC is not capable of real-time CDC because high frequency polling on a database resource will have a significant impact on the database's transaction rate. We also state that push CDC architectures are seen as the optimal solution for real-time CDC, as they do not poll the database. However, the issue with push CDC architectures are that they are intrusive towards current systems and difficult to maintain. Our TAAR solution to push CDC can minimise maintenance efforts, but it can still be intrusive towards current systems. Therefore, the discussion on real-time CDC weighed the impact of implementing the TAAR against the impact pull CDC's high frequency polling has on the database's transaction rate.

The outcome of the discussion can be summarised in Table 8.7, which presents the viability of each CDC technique at the various levels of concurrency. A tick indicates that the CDC mechanism can be used to provide real-time CDC at that concurrency level, whereas a cross means that the CDC mechanism is not capable of real-time CDC at a concurrency level.

Concurrency Level	**TAAR**	**Timestamp CDC**	**Trigger CDC**	**DBMS Log CDC**
Low	✔	✔	✔	✗
Medium	✔	✔	✗	✗
High	✔	✗	✗	✗

Table 8.7: Real-Time Viability at Varying Levels of Concurrency

The results suggested that at lower levels of concurrency pull CDC architectures that use timestamps or triggers can be used for real-time CDC. Moreover, at low levels of concurrency the results do not affirm our hypothesis, because pull CDC architectures are capable of real-time CDC. Push CDC is also capable of real-time CDC at low levels of concurrency. However, the impact of implementing a set TAAR from scratch may be greater than the impact pull CDC has on the database's transaction rate.

When the concurrency increases to medium and high levels, then the results do affirm our hypothesis. High frequency polling does have a significant impact on the transaction rate. To make pull CDC viable at these concurrency levels, one would have to add a sufficient amount of mechanical latency between successive poll operations. Such mechanical latency would mean that the capture latency in the pull CDC architectures would

no longer be comparable to push CDC. The cost of extra mechanical latency will be that changes are left undetected for longer, which means the value of the change data has longer to depreciate, because target systems will have to wait longer before they can do analysis on the data.

Chapter 9

Conclusions and Future Work

This chapter presents a conclusion of the work contained in this thesis. The purpose of this chapter is to tie together the various issues of the research that were covered in the body of the thesis, and reflect on the limitations of the work that has been done. Considering these limitations, the chapter moves on to discuss future work that may overcome these limitations. This chapter ends with some final thoughts regarding the thesis, and the subject area as a whole.

9.1 Aims

This section reviews the aim of this thesis, in the context of the four objectives that were set out in Chapter 1. The aim of this thesis was to:

- pragmatically develop a push CDC architecture and determine its viability as a real-time capture technology, by empirically evaluating it against pull CDC architectures.

To achieve this aim; the following research objectives were defined:

Objective 1 Design a push CDC architecture that will address the software engineering difficulties that exist in current push CDC architectures.

Objective 2 The proposed push CDC approach will not prevent a high transactional throughput on an OLTP database.

Objective 3 Design an experiment to benchmark the performance of CDC technologies.

Objective 4 Use the results obtained from the CDC benchmarks, to evaluate our push CDC architecture, by contrasting it to the performance of pull CDC architectures.

The following sections reflect on these objectives, and discusses how they were attained by the work in the thesis.

9.1.1 Objective 1

Chapter 2 put forward a theory that push CDC is capable of real-time change data capture whereas pull CDC is not. Moreover, we argued that to make pull CDC a real-time technology, one would have to poll a database resource at a high frequency to capture the data changes. High frequency polling would have an adverse affect on the database's transaction rate, which is undesirable. To allow for a high transaction rate, when using pull CDC one would have to increase the mechanical latency between successive poll operations. Increasing the mechanical latency would delay the capturing of change data and ultimately prevents pull CDC from being used to do real-time CDC. Push CDC mechanisms on the other hand, capture data changes on route to the database, and thus do not poll a DBMS resource to obtain the data changes. Therefore, a mechanical latency would not be required, which would mean push CDC is better suited for real-time CDC. The most common approach to push CDC is application based push CDC. The application that is responsible for sending data modification requests to the database will have first-hand knowledge of the changes that are being made to the database. CDC logic could be placed in the application to propagate changes on to target systems once they have been successfully applied to the database. The only latency in this system would be processing latency. Theoretically, push CDC is the optimal approach for minimising capture latency. A caveat with application based CDC is that it is highly intrusive towards current systems and is arduous to maintain, which means push CDC is seldom implemented. Therefore, the first objective of this thesis is to make a theoretical contribution to the CDC field, by developing a push CDC mechanism that is less intrusive towards current systems and is easier to maintain than existing solutions.

To solve the push CDC issues of intrusion and maintenance, Chapter 3 presented an SOA based CDC solution. Using the SOA principles, we were able to advance the theoretical knowledge of push CDC by proposing that one uses Data Access Services (DAS) for push CDC. SOA principles, implemented via DAS can improve push CDC architectures by allowing one to: decouple the CDC code from the application, so that applications are free from the responsibility of managing database communications; encapsulate the CDC logic behind a clean interface, so applications are protected from maintenance changes to the CDC logic; reuse CDC logic, so current and new applications do not have to replicate the logic.

9.1.2 Objective 2

Change data capture is most often applied to OLTP databases. For DAS CDC to be successful in an OLTP environment, the DAS must not place a significant overhead on the database transactions.

To attain this objective, Chapter 4 surveyed the DAS literature to discover DAS implementation methodologies. Two different methodologies to building DAS emerged: 1) generic DAS, which accept SQL and DML strings from the client applications and executes

these strings on the clients behalf; 2) schema specific DAS, where the services expose operations that are pertinent to a particular schema, and clients call these operations; passing in request parameters, and receiving a response from the service.

Chapter 3 theoretically discussed the implications each type of DAS methodology will have on the performance of a transaction in an OLTP environment. We concluded that a schema specific approach will provide optimal performance because an operation can expose a multi-action transaction, which will reduce the number of network round trips between the client and service. However, this was only theory, and our survey of the DAS literature found that previous performance studies focused heavily on generic DAS implementations, where the DAS were used to execute a single action through the DAS. We felt the need to supplement this literature with a study on DAS performance in an OLTP environment, and also provide empirical data on the performance of schema specific services.

We designed a set of preliminary experiments that evaluated the performance of different DAS implementations over a single transaction, which could vary in complexity. Our results showed that in an OLTP environment the schema specific services were more efficient for executing transactions. Additionally, we found that a RESTful architecture, slightly out performed SOAP based architectures. We labelled our schema specific approach as Transaction As A Resource (TAAR), because each service exposes a transaction as a resource.

On the whole we attained objective 2. However, placing the TAAR on the same server as the DBMS placed an additional workload on the server's CPU. We justify this approach by tying it in with current trends towards server centralisation, and utilisation of virtual servers. Although, to fully justify our approach would require us to either evaluate the system with different hardware configurations, or implement our approach in a production system that uses server virtualisation.

9.1.3 Objective 3

The third objective of this thesis is to develop a set of experiments to benchmark CDC performance. In chapter 6, we presented our benchmark. Our benchmark evaluated CDC technologies on three key performance measurements: transaction throughput, capture latency and CPU usage.

To attain this objective, we designed our benchmark around the TPC-C benchmark, because it is an industry recognised approach for gauging OLTP performance, and CDC often occurs on OTLP databases. For each CDC mechanism we were benchmarking we built a CDC system to capture the changes that are generated by the TPC-C benchmark. By measuring the TPC-C transaction rate (tpmC) when running different CDC configurations we were able to achieve our objective of determining the impact CDC has on the transaction rate.

To measure capture latency, we had to develop our own measurement process. A

standardised approach to measuring capture latency is undefined in the previous literature. Some works appear to measure only the time taken to identify changes, whilst others measure capture latency as the latency between the point where a change is extracted from the source database and the point where the change is loaded it into a data warehouse. We are concerned specifically with the amount of latency between a change occurring in the database and that change being captured; neither of the previous measurements was suitable here.

Our measurement process, measured capture latency as the time between two points: 1) the time at which the change is committed to the database; 2) the time at which the change is captured. To measure commit time we had to extend the TPC-C benchmark to add *transactionID* columns to the tables, which allowed us to tag each change with a *transactionID*, and with a *transactionId* a commit time. The *transactionId* was also tagged with a capture time, once the CDC mechanism has captured it. The capture time could then be calculated by subtracting the commit time from the capture time. Our tagging approach does not significantly affect the TPC-C benchmark or the DBMS performance.

Our definition of capture latency provides the first step towards standardising the measuring of capture latency. Our technique can be implemented by other researchers and practitioners in their own experiments. Furthermore, the capture latency measurement processes is independent to the TPC-C benchmark, so it can be fitted to any benchmark system. This is useful because others will be able to extend our work using systems that are more realistic than the TPC-C benchmark.

9.1.4 Objective 4

The fourth objective for this thesis was to use the results obtained from the CDC evaluation experiments, to enhance the field with a fresh interpretation on CDC technologies.

Our results showed that according to Definition 2, pull CDC architectures, were not capable of real-time CDC, because a 50ms mechanical latency had to be introduced into the pull CDC architectures. However, the results showed that at low levels of concurrency, a small amount of mechanical latency makes an insignificant difference to the capture latency in the system. In fact at low levels of concurrency our results showed that pull CDC architectures that are based on either timestamps or triggers, are capable of producing capture latencies that are comparable to the low capture latency observed in TAAR. Moreover, if one considers the amount of work a database can do in the additional time it takes pull architectures to capture changes, then there is not much difference between the TAAR solution and the pull CDC mechanisms. Furthermore, the impact of implementing DAS from scratch may mean that one may opt for a pull CDC mechanism for real-time CDC, when concurrency levels are low.

As the number of concurrent users scaled the TAAR CDC approach became more feasible because it could provide real-time CDC without impacting the database's transactional throughput. The TAAR could do this because its capture mechanism does not

compete with the transactional workload for database resources. At higher levels of concurrency the results affirmed our hypothesis that high frequency polling does prevent pull CDC architectures from being used for real-time CDC. The high frequency polling places an additional workload on the database, which competes with the transactional workload for resources. This competition for resources leads to both the degrading of transactional performance and an increase in capture latency.

Another way to consider our results is in terms of scaling. Our results show that in a real-time environment, push CDC architectures (TAAR CDC) scale better than pull CDC architectures. The TAAR CDC architecture scales better than pull CDC architectures because the TAAR change data identification and data capture mechanisms use system resources that are not in high demand by the transactional workload. Conversely the pull CDC mechanism identifies and captures change data by using resources that are in high demand by the transactional workload. Moreover, in a real-time environment the CDC workload from the pull CDC mechanisms directly conflict with the transactional workload. At medium and high levels of concurrency the conflict between the CDC workload and transactional workload, means that pull CDC architectures are no longer capable of doing real-time CDC.

9.2 Limitations

There were various limitations to the work in this thesis. This section highlights these limitations.

There are various limitations with TAAR based CDC. A possible limitation of TAAR based CDC is the assumption that all data modification will be done through the DAS. If applications modify the database through means other than the DAS, then potentially, new data will remain uncaptured. It is fine to enforce the policy of doing data modification via a DAS in a controlled academic environment. In a real-world scenario though, it may be more difficult to enforce. However, before we could deal with these kinds of issues, we felt it to be more important to investigate whether the DAS CDC performs well in an OLTP environment. After demonstrating that the proposal works, organisations can decide their own strategies for coping with emerging complexities.

Another limitation with TAAR based CDC, as presented in this thesis, is that we have only evaluated TAAR CDC from a performance perspective. This thesis has not provided an investigation into how TAAR CDC would integrate into a complete data warehouse system, such as Nguyen et al.'s architecture [78]. In theory, TAAR CDC could replace the sense services in Nguyen et al.'s SARESA (Sense and Respond Service Architecture) architecture. However, in practice, there may be hidden complexities that impact the practicality of the TAAR system.

Our preliminary experiment was limited in that it did not represent the complexities found in an OLTP environment. To better attain our objective, we evaluated the TAAR with the industry recognised TPC-C benchmark. The experiments could further be im-

proved if we evaluated TAAR CDC in an environment closer to a production environment. A production environment will contain complexities such security, additional background processes on the DBMS server, additional payloads on the network. These issues could influence the outcome of the results. Furthermore, we showed that our system, especially for pull CDC mechanism was restricted in terms of hardware performance. A server equipped faster hard disks speeds, would likely have seen the pull CDC architectures perform better.

This thesis has not considered how the TAAR would work in a near real-time environment. The TAAR generates copious amounts of data in a short period of time, which may overwhelm a near real-time system. To prevent near real-time systems from being overwhelmed with change data, solutions are required to temporarily hold change data until it is needed. These solutions can already be found in the literature. For example Wang and Liu's [107] data warehouse cache loading system could be used to prioritise change data. The change data that is needed immediately can be placed in a cache for real-time data. The change data that is not needed immediately can be stored in caches that are set up for systems that need the change data at a later time. Another example is Karakasidis et al.'s [60] queue based system for loading a data warehouse. In a near real-time architecture, a queue system could be used to buffer change data until the target require it. A second solution is Kimball's [62] real-time partition. The change data could be stored in a real-time partition, and switched over to the static warehouse partition when the targets actually need to use the change data.

Finally, this thesis has not considered how the TAAR would scale in a real-world scenario. Highleyman [50] discuss a fraud analysis case study where CDC is applied to a banking system that can have a transactional throughput of up to 5000 transactions per second. The results in this thesis demonstrated that the TAAR are suitable for real-time CDC when the concurrency is high, and a high transactional throughput is required. However, our results showed that the TAAR approach will eventually become CPU bound. It is likely that when using our hardware 5000 transactions per second will lead the TAAR to thrash the CPU resources. A possible solution to additional scaling can come from the centralised computing approach we have taken with the TAAR. It is possible that one will be able to scale the TAAR solution horizontally, by adding more CPU cores to the server when they are required. Adding more CPU will possibly allow the TAAR to continue to serve the both the CDC logic and the transactional logic. However, an evaluation of the TAAR in a real-world scenario is required, to discover feasible this would be.

9.3 Future Work

There are numerous avenues of future work in CDC field. This section presents a limited set of these avenues here.

9.3.1 Extending the TAAR CDC Architecture

The work in this thesis has only considered the change capture part of a CDC system, which allowed us to focus primarily on CDC performance. However, capturing data is only one feature of a CDC system, other features include: change delivery; transactional consistency; change staging; change metadata population. Furthermore, as shown by Nguyen et al. [78], CDC itself is part of a wider architecture, where the role of CDC is to feed other systems with fresh change data. Future work could investigate how TAAR CDC may integrate with a complete real-time data warehouse architecture. Such an investigation, in the form of case-studies, could potentially help to discover whether TAAR CDC works in practice, and not just in an academic study on performance.

9.3.2 Exploring the Intrusiveness of TAAR CDC

Our investigation into capture latency would be most beneficial if coupled with a study on CDC intrusiveness. This thesis developed the idea of improving push CDC by lessening the impact of intrusiveness and the maintenance burden through the software engineering principles of SOAs. Now that we have shown the TAAR concept to works from a performance perspective it would be ideal to complement this study with a study that measures how intrusive each CDC mechanism is. An intrusiveness study will remove uncertainty and will provide practitioners a more comprehensive overview of CDC, which will help them decide the optimal CDC mechanism for their environment. Such a study would have to define a metric for intrusiveness and determine the optimal way to measure it. It is likely that such a study would be best conducted on a real-world scenario, because then one can gain a full appreciation for how a system will need to change to incorporate push CDC.

9.3.3 Extending the CDC Investigation into Capture Time Latency

The TPC-C OLTP system used in this thesis is an abstract and simplified model of an OLTP environment. The TPC-C benchmark includes only five transactions, which is far removed from the complexity of reality. Our study can be enriched with case studies on real-world implementations of TAAR services that encapsulate the OLTP database for CDC purposes. A real-world environment is likely to have many more transactions, which would require many more services. This thesis does not fully consider the resources and infrastructure required to support the TAAR methodology on a larger scale. Real-world issues like load balancing, security and usage policies can all affect either performance or the TAAR's ability to fully function as a CDC mechanism. Furthermore, the hardware in our DBMS server is unlikely to match up to an organization's production system. Therefore, real-world case studies will address the limitations of this research, and uncover hidden complexities that may have been overlooked in this study.

Finally, this thesis did not measure capture latency that is caused by the commercial

external log CDC tools. External log CDC is purported to be capable of sub second capture latency. However, this has not been proven with a study as comprehensive as the one in this thesis. Furthermore, it is possible that previous studies have not measured the time to fully capture a change, and instead only measured the time to identify a change. We have extended the TPC-C benchmark to measure capture latency, which includes full capture of a record. Our approach to measuring capture latency is simple enough to be implemented in any OTLP benchmark. We hope practitioners and external log CDC vendors extend our work by measuring the capture latency that is caused by their external log CDC tools.

9.4 Final Thoughts

I hope that the work in this thesis will provide a platform to open up a lively discussion on capture latency in CDC systems. Furthermore, I hope that this study will encourage our peers in both academia and in industry to enrich our data with results from their own experiments.

References

[1] S. Abiteboul, O. Benjelloun, and T. Milo. Web services and data integration. In *Web Information Systems Engineering, 2002. WISE 2002. Proceedings of the Third International Conference on*, pages 3 – 6, December 2002. 50

[2] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 882–884, New York, NY, USA, 2005. ACM. 25

[3] Mohammad Alrifai, Peter Dolog, Wolf-Tilo Balke, and Wolfgang Nejdl. Distributed management of concurrent web service transactions. *IEEE Transactions on Services Computing*, 2:289–302, 2009. 61

[4] Amazon. Amazon simple storage service (amazon s3). http://aws.amazon.com/s3/. 54

[5] Scott Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, NY, USA, 2003. 49, 61

[6] Ali Anjomshoaa, Mario Antonioletti, Malcolm Atkinson, Rob Baxter, Andrew Borley, Neil P. Chue Hong, Brian Collins, Neil Hardman, Alastair C. Hume, Alan Knox, Mike Jackson, Amy Krause, Simon Laws, James Magowan, Norman W. Paton, Dave Pearson, Tom Sugden, Paul Watson, and Martin Westhead. The design and implementation of grid database services in ogsa-dai: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):357–376, 2005. 49, 50

[7] Mario Antonioletti, Malcolm P. Atkinson, Robert M. Baxter, Andrew Borley, Neil P. Chue Hong, Brian Collins, Neil Hardman, Alastair C. Hume, Alan Knox, Mike Jackson, Amrey Krause, Simon Laws, James Magowan, and Pato. The design and implementation of grid database services in ogsa-dai. *Concurrency - Practice and Experience*, 17(2-4):357–376, 2005. 49

[8] Attunity. Efficient and real time data integration with change data capture. White Paper, 2009. 14, 26, 31, 32, 34, 41, 44

[9] C.K Bhensdadia and Yogeshwar.P Kosta. Empirical study on dynamic warehousing. *International Journal of Computer Theory and Engineering*, 2:751–759, 2010. 19

[10] Vinayak R. Borkar, Michael J. Carey, N. Mangtani, D. McKinney, R. Patel, and Sachin Thatte. Xml data services. *Int. J. Web Service Res.*, 3(1):85–95, 2006. 49, 56

[11] Karoly Bosa and Wolfgang Schreiner. A Grid Software for Virtual Eye Surgery Based on Globus 4 and gLite, May 2007. 49

[12] Mokrane Bouzeghoub. A framework for analysis of data freshness. In *Proceedings of the 2004 international workshop on Information quality in information systems*, IQIS '04, pages 59–67, New York, NY, USA, 2004. ACM. 10, 84

[13] Robert M. Bruckner, Beate List, and Josef Schiefer. Striving towards near real-time data integration for data warehouses. In *DaWaK 2000: Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery*, pages 317–326, London, UK, 2002. Springer-Verlag. 18, 22

[14] D Burleson. New developments in oracle data warehousing. Online. Retrieved October 14, 209, from the World Wide Web: http:www.dba-oracle.comoracle_news2004_4_22_burleson.htm, 2010. 25

[15] D Burleson. A glimpse into the future of database management. DBAZine. Retrieved June 23, 2011, from the World Wide Web: http://www.dba-oracle.com/art_dbazine_2020_p2.htm, 2010. 38, 80

[16] D Burleson. Oracle metric log file switch completion. DBAZine. Retrieved June 23, 2011, from the World Wide Web: http:www.dba-oracle.comm_log_file_switch_completion.htm. 97

[17] Junwei Cao, S.A. Jarvis, S. Saini, and G.R. Nudd. Gridflow: workflow management for grid computing. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 198 – 205, may 2003. 25

[18] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10(2-3):199–223, September 2001. 25

[19] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, 1997. 25

[20] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah, and Jianyong Wang. Multi-dimensional regression analysis of time-series data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 323–334. VLDB Endowment, 2002. 26

[21] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003. 25

[22] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the limits of soap performance for scientific computing. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 246, Washington, DC, USA, 2002. IEEE Computer Society. 63

[23] S. Cholia, D. Skinner, and J. Boverhof. Newt: A restful service for building high performance computing web applications. In *Gateway Computing Environments Workshop (GCE), 2010*, pages 1 –11, November 2010. 54, 60

[24] Munir Cochinwala and Euthimios Panagos. Near realtime call detail record etl flows. In Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Malu Castellanos, Umeshwar Dayal, and Rene J. Miller, editors, *Enabling Real-Time Business Intelligence*, volume 41 of *Lecture Notes in Business Information Processing*, pages 142–154. Springer Berlin Heidelberg, 2010. 18, 19, 21, 126

[25] S.S. Conn. Oltp and olap data integration: a review of feasible implementation methods and architectures for real time data analysis. In *SoutheastCon, 2005. Proceedings. IEEE*, pages 515 – 520, april 2005. 39

[26] Transaction Processing Performance Council. TPC-C. http://www.tpc.org/tpcc/spec/tpcc_current.pdf, 2010. 16, 71, 74, 75, 77

[27] Szepielak Daniel, Tumidajewicz Przemyslaw, and Hagge Lars. Integrating information systems using web oriented integration architecture and restful web services. In *SERVICES '10: Proceedings of the 2010 6th World Congress on Services*, pages 598–605, Washington, DC, USA, 2010. IEEE Computer Society. 54, 61

[28] Umeshwar Dayal, Malu Castellanos, Alkis Simitsis, and Kevin Wilkinson. Data integration flows for business intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 1–11, New York, NY, USA, 2009. ACM. 25

[29] Yanbo Deng. Using web services for customised data entry. Master's thesis, Lincoln University, 2007. 50, 55, 58, 59, 65, 74

[30] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 61–72, New York, NY, USA, 2002. ACM. 25

[31] Erdogan Dogdu. A generic database web service. In *in Proc. of the 2006 Int. Conf. On Semantic Web and Web Services*, 2006. 49, 55, 65, 74

[32] eBay. ebay shopping apis. http://developer.ebay.com/products/shopping/. 54

[33] Mitchell J. Eccles, David J. Evans, and Anthony J. Beaumont. True real-time change data capture with web service database encapsulation. *Services, IEEE Congress on*, 0:128–131, 2010. 27

[34] Wayne W. Eckerson. Best practices in operational bi. converging analytical and operational processes. Tdwi best practices report, TDWI, 2007. 20, 21, 126

[35] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, August 2005. 42, 48

[36] eti. Simplifying data management with change data capture. White paper available at www.eti.com/white_papers/wp_ETI_CDC.pdf. 41

[37] Evans Data Corporation. Database development survey. *Data Management and DataBase Trends*, 1, 2004. 53

[38] J. Roberto Evaristo, Kevin C. Desouza, and Kevin Hollister. Centralization momentum: the pendulum swings back again. *Commun. ACM*, 48:66–71, February 2005. 80

[39] Facebook. Graph api. http://developers.facebook.com/docs/reference/api/. 54

[40] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. 61

[41] Flex CDC. Flexcdc. Technical Document available at: http://code.google.com/p/flexviews/wiki/FlexCDC, March 2011. 32

[42] Rajesh Gadodia. Right in time: to survive today's challenges, businesses are taking a long look at their data warehousing investment–and wondering how to break traditional barriers, move information faster, and serve a broader class of users. *Intelligent Enterprise*, 7, 2004. 21

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. 23

[44] D. Geer. Will binary xml speed network traffic? *Computer*, 38(4):16 – 18, april 2005. 60

[45] Rick Greenwald, Robert Stackowiak, and Jonathan Stern. *Oracle essentials, 3rd edition*. O'Reilly, third edition, 2004. 36, 97, 118

[46] Rick Greenwald, Robert Stackowiak, and Jonathan Stern. *Oracle essentials, 3rd edition*. O'Reilly, third edition, 2004. 54, 57, 58

[47] Joseph Guerra and David Andrews. Creating a real time data warehouse. White Paper available at www.rapiddecision.net/pdfs/Creating_RealTime_DW.pdf, 2011. 22

[48] Richard Hackathorn. Active dta warehouising: From nice to necessary. *Teradata Magazine*, 6, 2006. 8, 19, 126

[49] Mark D. Hansen, Stuart E. Madnick, and Michael Siegel. Data integration using web services. *SSRN eLibrary*, 2002. 50

[50] Bill Highleyman. Shadowbase streams for application integration. White Paper, August 2011. 31, 32, 36, 134

[51] HiTSoftware. Real-time data replication and change data capture. White Paper, July 2011. 31, 32

[52] HVR Software. Hvr realtime data integration. White Paper, 10 2009. 31, 32, 33, 34, 38, 39, 44, 84, 124, 125

[53] IBM. Help alleviate batch windows with timely, reliable, data delivery. An IBM white paper available at, http://www-01.ibm.com/software/data/infosphere/change-data-capture/system-z/, September 2010. 26, 31, 32, 44

[54] William H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., New York, NY, USA, 1992. 18

[55] JSON.org. http://json.org/xml.html. 60

[56] Liu Jun, Hu ChaoJu, and Yuan HeJin. Application of web services on the real-time data warehouse technology. In *Advances in Energy Engineering (ICAEE), 2010 International Conference on*, pages 335 –338, june 2010. 43

[57] Wang Junye, Mao Lirui, and Cai Hongming. A rest-based approach to integrate enterprise resources. In *Computer Science-Technology and Applications, 2009. IFCSTA '09. International Forum on*, volume 3, pages 219 –223, 25-27 2009. 54

[58] Thomas Jörg and Stefan Dessloch. Towards generating etl processes for incremental loading. In *Proceedings of the 2008 international symposium on Database engineering and applications*, IDEAS '08, pages 101–110, New York, NY, USA, 2008. ACM. 35

[59] Thomas Jörg and Stefan Dessloch. Near real-time data warehousing using state-of-the-art etl tools. In Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Malu Castellanos, Umeshwar Dayal, and Rene J. Miller, editors, *Enabling Real-Time Business Intelligence*, volume 41 of *Lecture Notes in Business Information Processing*, pages 100–117. Springer Berlin Heidelberg, 2010. 18, 21

[60] Alexandros Karakasidis, Panos Vassiliadis, and Evaggelia Pitoura. Etl queues for active data warehousing. In *Proceedings of the 2nd international workshop on Information quality in information systems*, IQIS '05, pages 28–39, New York, NY, USA, 2005. ACM. 25, 134

[61] Toshiaki Katayama, Mitsuteru Nakao, and Toshihisa Takagi. Togows: integrated soap and rest apis for interoperable bioinformatics web services. *Nucleic Acids Research*, 38(suppl 2):W706–W711, 2010. 55, 61

[62] Ralph Kimball. Kimball design tip #31: Designing a real time partition. Online at: http:www.kimballgroup.comhtmldesigntipsPDFDesignTips2001KimballDT31Designing.pdf, November 2001. 25, 134

[63] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleanin.* John Wiley & Sons, 2004. 12, 21, 35, 40, 124

[64] Michael Koch, Markus Hillenbrand, and Paul Muller. A generic database web service for the venice service grid. In *AINA '09: Proceedings of the 2009 International Conference on Advanced Information Networking and Applications*, pages 129–136, Washington, DC, USA, 2009. IEEE Computer Society. 49, 50, 55, 60, 65, 74

[65] Paul Lane. *Oracle Database Data Warehousing Guide*, chapter Chapter 16, pages 199–271. Online: http:docs.oracle.comcd/B28359_01server.111b28313.pdf, 2007. 29, 33

[66] Justin Langseth. Real-time data warehousing challenges and solutions. Article published at DSSResources.COM, February 2004. 19, 21, 25

[67] Wolfgang Lehner and Wolfgang Hmmer. The revolution ahead: Publishsubscribe meets database systems, 2001. 23

[68] Y. Liu and I. Gorton. An empirical evaluation of architectural alternatives for j2ee and web services. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 10 – 17, nov.-3 dec. 2004. 65

[69] John Loffink. Exploring the dell poweredge m1000e network fabric architecture. Dell Power Solutions, February 2008. 80

[70] Xinjie Lu, Xin Li, Tian Yang, Zaifei Liao, Wei Liu, and Hongan Wang. Qos-aware publish-subscribe service for real-time data acquisition. In Malu Castellanos, Umesh Dayal, Timos Sellis, Wil Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, and Clemens Szyperski, editors, *Business Intelligence for the Real-Time Enterprise*, volume 27 of *Lecture Notes in Business Information Processing*, pages 29–44. Springer Berlin Heidelberg, 2009. 23

[71] Denis Lussier. Benchmarksql-2.3.2. http://benchmarksql.sourceforge.net/index.html, 2006. 75

[72] Michael V. Mannino and Zhiping Walter. A framework for data warehouse refresh policies. *Decis. Support Syst.*, 42:121–143, October 2006. 18

[73] Microsoft. Basics of change data capture. Available online at http:msdn.microsoft.comen-uslibrarycc645937.aspx, November 2009. 29, 30

[74] Microsoft Corporation. Web services performance: Comparing java 2tm enterprise edition (j2eetm platform) and the microsoft .net framework a response to sun microsystem's benchmark. White Paper, July 2004. 64, 65

[75] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965. 80

[76] Gary Myers. Recording the commit time on a record. Retrieved February, 2011, from the World Wide Web: http://blog.sydoracle.com/2005/10/recording-commit-time-on-record.html, October 2005. 86

[77] Ian Naumann, Emma Lumb, Kerry Taylor, Robert Power, David Ratcliffe, and Michael Kearney. The australian plant pest database: A national resource for an expert community. In *In Proceedings The Twelfth Australasian World Wide Web Conference*, 06. 50

[78] Tho Manh Nguyen, Josef Schiefer, and A. Min Tjoa. Sense & response service architecture (saresa): an approach towards a real-time business intelligence solution and its use for a fraud detection application. In *Proceedings of the 8th ACM international workshop on Data warehousing and OLAP*, DOLAP '05, pages 77–86, New York, NY, USA, 2005. ACM. 19, 24, 25, 43, 126, 133, 135

[79] Tho Manh Nguyen, A Min Tjoa, Edgar Weippl, and Peter Brezany. Towards a grid-based zero-latency data warehousing implementation for continuous data streams processing. *International Journal of Data Warehousing and Mining*, 1(4):22, 2005. 25

[80] William Norcot. Synchronous change data capture in a relational database, May 2001. US Patent No. US7111023 26, 30, 33

[81] William Norcot, Michael Brey, John Galanes, Paula Bingham, and Raymond Guzman. Method and apparatus for change data capture in a database system, May 2002. US Patent No. US6999977 12, 14, 30, 34, 35, 37, 72, 124

[82] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. *Scenario*, 59715:157–162, 2009. 73

[83] OASIS. Web service atomic transaction (ws-atomictransaction). http://docs.oasis-open.org/ws-tx/wsat-1.1-spec-os.pdf, 2007. 61

[84] OASIS. Web service business activity (ws-businessactivity). http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec.pdf, 2007. 61

[85] Oracle. Oracle goldengate 11g: Real-time access to real-time information. White Paper, 02 2012. 31, 32, 84

[86] V. F. Pais, V. Stancalie, F. A. Mihailescu, and M. C. Totolici. Performance issues related to web service usage for remote data access. *AIP Conference Proceedings*, 996(1):276–280, 2008. 65

[87] V.F. Pais and V. Stancalie. Using web services for remote data access and distributed applications. *Fusion Engineering and Design*, 81(15-17):2013 – 2017, 2006. 5th IAEA TM on Control, Data Acquisition, and Remote Participation for Fusion Research - 5th IAEA TM. 50

[88] Alok Pareek. Addressing bi transactional flows in the real-time enterprise using goldengate tdm. In Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Malu Castellanos, Umeshwar Dayal, and Rene J. Miller, editors, *Enabling Real-Time Business Intelligence*, volume 41 of *Lecture Notes in Business Information Processing*, pages 118–141. Springer Berlin Heidelberg, 2010. 22, 26, 30, 38, 41, 48, 86, 124, 125

[89] Damir Pintar, Mihaela Vranić, and Zoran Skočir. Metadata-driven soa-based application for facilitation of real-time data warehousing. In *EC-Web 2009: Proceedings of the 10th International Conference on E-Commerce and Web Technologies*, pages 108–119, Berlin, Heidelberg, 2009. Springer-Verlag. 43, 54

[90] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.E. Frantzell. Supporting streaming updates in an active data warehouse. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 476 –485, april 2007. 25

[91] QuestSoftware. Shareplex for oracle. White Paper, September 2009. 31, 32

[92] Jack Raitto. Oracle change data capture. Presentation, available at http://nyoug.org/Presentations/SIG/LI/lisigcdc.ppt, October 2004. 39, 83, 84, 125

[93] Raghu Ramakrishnan, Johannes Gehrke, Raghu Ramakrishnan, and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 3 edition, August 2002. 36, 118

[94] Denis Raphaely. *Oracle Streams*. Online: http://docs.oracle.com/cd/B28359_01/server.111/b28420.pdf, 2007. 30

[95] B.D Reimers. Real-time data: Too much of a good thing? Computer World. Retrieved June 07, 2010, from the World Wide Web: http://www.computerworld.com/s/article/80203/Too_Much_of_a_Good_Thing_, 2003. 18, 19, 20, 21

[96] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, 1 edition, 2007. 54, 61

[97] Lutz Schlesinger, Florian Irmert, and Wolfgang Lehner. Supporting the ETL-process by web service technologies. *Int. J. Web Grid Serv.*, 1:31–47, August 2005. 44

[98] Sense Corp. Data integrator: Change data capture with database triggers. White paper, available from http://www.businessobjects.com/pdf/dev_zone/di_triggers_wp.pdf, 2004. 33

[99] JinGang Shi, YuBin Bao, FangLing Leng, and Ge Yu. Study on log-based change data capture and handling mechanism in real-time data warehouse. In *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, pages 478–481, Washington, DC, USA, 2008. IEEE Computer Society. 31, 33

[100] Bongsik Shin. An exploratory investigation of system success factors in data warehousing. *Journal of the Association for Information Systems*, 4(6):141–170, 2003. 18

[101] Sun Microsystems Inc. Web services performance comparing javatm 2 enterprise edition (j2eetm platform) and .net framework. White Paper, June 2004. 64

[102] Simon Terr. Real-time data warehousing 101. Retrieved May 31, 2010, from the World Wide Web: http://www.information-management.com/news/7524-1.html, October 2003. 18, 20, 21

[103] Dimitri Theodoratos and Mokrane Bouzeghoub. Data currency quality factors in data warehouse design. In *In Proc. of the Int. Workshop on Design and Management of Data Warehouses (DMDW'99*, 1999. 84

[104] C. Thomsen, T.B. Pedersen, and W. Lehner. Rite: Providing on-demand data for right-time data warehousing. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 456 –465, april 2008. 25

[105] Panos Vassiliadis and Alkis Simitsis. Near real time etl. In *New Trends in Data Warehousing and Data Analysis*, pages 1–31. 2009. 25

[106] K. Velmurugan and M.A.M. Mohamed. An empirical performance metrics measurement and analysis of software platforms for implementation of web services. In *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, pages 914 –920, 1-3 2009. 65

[107] Cuiru Wang and Shuangxi Liu. Soa based electric power real-time data warehouse. In *Proceedings of the 2008 Workshop on Power Electronics and Intelligent Transportation System*, PEITS '08, pages 355–359, Washington, DC, USA, 2008. IEEE Computer Society. 26, 42, 43, 134

[108] Richard Y. Wang and Diane M. Strong. Beyond accuracy: what data quality means to data consumers. *J. Manage. Inf. Syst.*, 12:5–33, March 1996. 18

[109] Wisdom Force. Wisdomforce dbsync datasheet. White paper, 05 2011. 31, 32

[110] Hoschek Wolfgang and Gavin McCance. Grid enabled relational database middleware. Presented at the Global Grid Forum, Frascatti, Italy, 2001. 61

[111] Youchan Zhu, Lei An, and Shuangxi Liu. Data updating and query in real-time data warehouse system. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 5, pages 1295 –1297, December 2008. 26, 43