Concurrency Control in Database Management Systems

Survey Paper for CSC443

Alec Brunelle
Department of Computer Science
University of Toronto, St.George
Toronto, Canada
alec@alec.coffee

Hilal Dib
Department of Computer Science
University of Toronto, St.George
Toronto, Canada
hilaldib@gmail.com

Spencer Elliott
Department of Computer Science
University of Toronto, St. George
Toronto, Canada
me@elliottsj.com

***Abstract*—With the rising need for applications that handle thousands or even millions of users at once, concurrency of database transactions is becoming increasingly important to ensure databases can maintain data integrity while efficiently handling the high volume of simultaneous reads and writes. This paper surveys multiple concurrency control techniques which are designed to increase the concurrency of database transactions while maintaining the correctness and consistency of the database. Specific problems and common concurrency control methods are described with conclusions being drawn into their efficient use cases. Trends such as concurrency control related to distributed database systems and mobile databases are also described.**

Keywords—concurrency control; relational databases; database management systems; ACID; distributed databases; mobile databases;

Introduction

SQL commands are grouped into what are called *transactions*, which are logical, atomic units of work which can be committed, rolled back and undone onto a database [2]. This grouping helps the database management system maintain certain properties that enable it to store data correctly and efficiently with support for fault tolerance. The basic properties of a transaction are *atomicity*, *consistency*, *isolation*, and *durability*, commonly abbreviated as *ACID*. This ensures that operations within a transaction are done all at once, transactions follow schema constraints, multiple transactions have the appearance of being done serially, and transaction results are persisted, even in the event of system failure. When transactions are performed sequentially, these basic properties can be upheld without much complexity [8]; concurrent transactions on the other hand require more consideration. Concurrency can be defined as the ability for multiple processes to access or change shared data at the same time [3]. A database wants to be able to support concurrent transactions to increase average throughput to individual end-users and complete jobs faster. An example of concurrency in a database management system occurs when two transactions want to access the same data at the same time, where transaction one has a write query on the same selection of data which transaction two is writing to. To ensure *ACID* and other properties, the database must not let two transactions mutate the same data at the same time, impacting concurrency [3]. The potential for failure of *ACID* properties, and thus failure of overall correctness, increases as more concurrent transactions begin to be performed by the database. The need for concurrency control is further increased by the recent popularity of distributed database systems, databases which reside on multiple machines and communicate through networks. Methodologies need to be created for the database to be able to handle concurrent transactions successfully. Several well known problems related to database concurrency control exist, including: lost updates, dirty reads, inconsistent analysis, phantom reads and missing/double reads [4]. With these problems come solutions, many popular formalized control methods exist: *locking(strict/non-strict two-phase)*, *timestamp ordering* and *commitment ordering*. *Integrity* and *serializability* are the two main factors of correctness which these methods must uphold [5]. We will use the following attributes as our classification scheme to classify different concurrency control techniques: level of correctness, performance overhead, and complexity. Formal examples of performance evaluation related to concurrency that we will be using are: transactional response time, throughput of transactions and rollback/blocking mechanisms. Concurrency control methods described in this survey are the result of many years of research and are used as the building blocks when developing new concurrency control methods.

This paper surveys, summarizes, and assesses topics related to problems which could arise when concurrent transactions are performed, methods used to solve these problems and properties of correctness these methods try to uphold. We also discuss the relationship between distributed databases and concurrency control, and methods to increase concurrency with long-lived transactions, as well as topics for future research.

I. Concurrency Effects

Before discussing the solutions to concurrency control in database management systems, we first must understand the associated problems. When performing transactions concurrently, we may run into undesirable results. For databases to maintain consistent results, they must account for these common problems.

A. The Lost Update Problem

The Lost Update Problem is described as the scenario in which two transactions grab the same selection of data and then update the data based on the values originally selected [4]. Whoever writes the data last will overwrite anything the other transaction wrote. A simple solution to this would be that the second transaction is not allowed to read data until the first has finished writing. This is described as a lock mechanism and is later described in more detail.

B. The Dirty Read Problem

Database management systems must support abortion of transactions. This requirement is set by the need for safe rollbacks in the occurrence of system failure and other types of failure. Transactions must perform atomic mutations upon a database, this means when another process or outside force needs to abort a transaction, it must be able to abort safely. *The Dirty Read Problem* occurs when concurrent transactions are being done without considering the atomic requirement. An instance of when this problem may occur is when a transaction is reading data that was written by a concurrent transaction which was later aborted [4]. A specific lock-based concurrency control method which is abort-orientated is the Basic Aborting Protocol (BAP). This method defines priorities for transactions and aborts low priority transactions for high priority ones as it attempts to increase timeliness of data accesses and throughput [5]. Concurrency control methods such as BAP must consider *The Dirty Read Problem* and ensure atomicity is achieved.

C. The Non-repeatable Reads Problem

This problem is described to occur when two identical reads are performed in the same transaction and one result is different from the other [4]. A specific example of when this could occur is when a transaction needs to read the same row from the database twice before committing. Before the second read operation is performed, a concurrent transaction could change data pertaining to the previously read row. Isolation is clearly violated as a single transaction cannot know anything about another incomplete transaction.

Aside: Oracle Database example isolation methods related to The Non-Repeatable Reads Problem

The Oracle Database is a popular implementation of a database management system which offers two different isolation levels, both differing in side-effects pertaining to *The Non-repeatable Reads Problem*. The choice between the two depends on expected transaction arrival rates and response times.

Read Committed Isolation is one isolation method which sacrifices the guarantee for non-repeatable reads to no occur for increased throughput and faster response times [6]. This isolation level gives transactions consistency for every query but does not guarantee consistency for the entire transaction itself. This method assumes that applications which have a low transaction arrival rate have a low risk of being the victim of *The Non-Repeatable Read Problem*. Another extension of this assumption is that databases do not see many transactions reading the same data twice on a query-by-query level.

The second method, *Serializer Isolation*, provides consistency at the transaction level. Having transaction-level consistency means subsequent reads will always be the same if it is in the same transaction [6]. Not only does this prevent issues related to *The Non-Repeatable Read Problem*, but also *The Dirty Read* and *Phantom Read Problems*. This option seems the most appealing for databases where many of the transactions are relatively short. An unfortunate consequence incurred from the usage of this method is that the application interacting with the database needs to handle concurrent update errors. An error like this occurs when a transaction tries to update or delete data changed by another transaction that committed after the first transaction began.

II. Traditional Concurrency Control Synchronization Methods

Many methods have been researched extensively trying to solve concurrency control within database management systems. A common trait between each system is that they

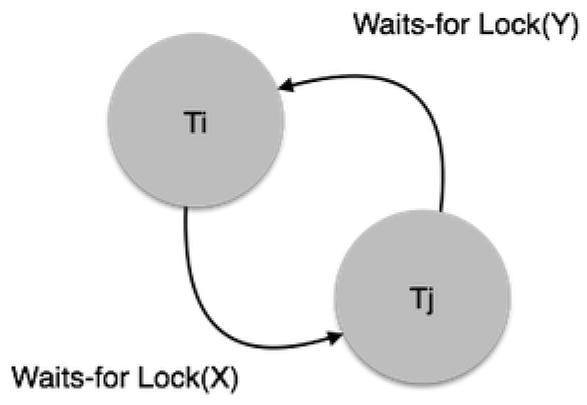

Fig. 2: Example Deadlock

must ensure *integrity* and *serializability*. *Synchronization* is a term used to define how transactions work in tandem with each other. Three general approaches have been assessed in research: *Wait*, *Timestamp*, and *Rollback* [8].

A. Wait

When two transactions need to read/write the same set of data at the same time, *Wait* methods simply make transactions wait upon each other. This makes sure concurrent transactions adhere to ACID properties and no conflicts arise. A very common way in which *Wait* concurrency control methods achieve this is with lock-mechanisms. Locks can attach themselves to sets of data such as tables, indexes or rows. This attaching process and also the amount of waiting that transactions might perform is a cause of overheard in this method. Locks can be segmented into different categories such as read and write or shared and exclusive to reduce the time transaction wait on each other [8]. *Wait* methods which use locks also have their own set of problems, mainly *Livelock* and *Deadlock*. *Livelock* is when transactions continually try to obtain an entity for locking and never obtain it due to other locks already being on the data.. *Deadlock* happens when multiple transactions obtain locks which depend on each other. Bhargava [8] has observed that deadlock resolution should be focused on more than deadlock prevention as transactional deadlock is rare. Figure 2 shows a simple example of two transactions when in deadlock.

A popular lock-based concurrency model which assures serializability is Two-Phase Locking (2PL). This model has two phases, the growing phase and the shrinking phase. During the growing phase, an increasing number of locks are obtained, and during the shrinking phase these locks are released. A variation of 2PL is the wound-wait method, the primary difference being the way deadlocks are handled. Rather than maintaining waits for information and checking for deadlocks, wound-wait uses timestamps. If transaction T_i requests a lock on a data item held by T_j, two outcomes are possible:

1. TS(T_i) < TS(T_j): T_i is older than T_j, so T_j is aborted and forced to be rolled back.
2. TS(T_i) > TS(T_j): T_i is newer than T_j, so T_i is forced to wait until T_j releases the lock.

B. Timestamp

In *Timestamp* ordering each transaction is assigned a unique timestamp. Timestamps are inherently unique and must maintain this even in a distributed environment. After assignments are done, serialization order when conflicts arise is simply to order the transactions temporally [8]. Two subsets methods of *Timestamp* ordering are Timestamp ordering with Transaction Classes and the Distributing Voting Algorithm [8].

C. Rollback

As locking and timestamp models preprocess information to make sure conflicts do not arise, rollback mechanisms take a more relaxed approach. The idea of rollback concurrency models are to let transactions perform actions and then validate those actions they performed after the fact. This model is often defined as *optimistic* as it assumes no conflicts exist at the beginning. Only when a validation phase fails, do transactions rollback and try validation once more. The overhead incurred by this method is in the amount of aborts which may be caused during the validation phase. Optimistic concurrency control methods have four phases, listed in order: *Read*, *Compute*, *Validate* and *Commit/Write [8]*. The *Read* phase simply reads values which are requested from the transaction. Because no loss of integrity can happen here, the values read are saved locally with validation occurring in later phases if conflicts arise. The *Compute* phase looks at the write related actions in the transaction and finds out what entities in the database will be a part of a write set. The computation done here is saved locally as we have not completed the

Validation phase yet. Read and Write sets were described earlier to be saved locally, and this is intended for use when validating against other concurrent transactions. The set of previously committed transactions cannot change during a transactions validation phase.

A notable rollback method is *multiversion concurrency control* (MCC). In MCC, when a read is made to the database a *snapshot* is taken of the database at the given moment. The advantage of MCCs is that there is no waiting for locks, however they can come with a significant space overhead.

D. Performance Analysis of Concurrency Control Methods

Two main ideas are presented to measure the degree of efficiency when performing analysis on concurrency control methods: *Degree of Concurrency* and *System Behaviour*.

Degree of Concurrency

An example of a low degree of concurrency is if a database performs all transactions serially. If *Two-Phase Locking* or *Optimistic Concurrency Control* are used, this demonstrates a high degree of concurrency. The degree of concurrency is a very general broad term.

System Behaviour

A more in-depth analysis into concurrency control methods can be attained when looking at system behaviour. System behaviour of transactions is described as response times, throughput and rollback/blocking. Many researchers have put together papers explaining performance differences between the different concurrency methods we have described so far. As they tend to describe specific scenarios, we will attempt to describe them in a more general sense. Comparing *Wait* and *Rollback* based methods, Rollback based methods perform considerably better when there is a mix of small and large transactions [8]. A low amount of conflicts are in fact the norm, with figure 1 giving evidence.. Let M be the amount of tuples in the database and we assign B to be the average size of a read/write set.

Bhargava presents a way to represent combinations for choosing B objects from a set of M objects [8]. Given these variables we can derive the maximum probability these two values will data a single data object.

$$1 - \frac{(M - 2B + 1)^B}{(M - B + 1)}$$

We now give an example from a data set representing the chance of conflict.

B	*M*	*Chance of Conflicts*
5	100	0.0576
10	500	0.0025
20	1000	0.1130

Fig. 1. Sample inputs into conflict equation [8]

Given that a low amount of conflicts is quite common, studies found that when using *Rollback*-based approaches there were fewer aborts, but when using *Wait*-based approaches, there were less blocking. In the event of a high amount of conflicts, both of these methods decrease in execution time similarly [8]. Using this data we can infer that *Optimistic* approaches will be more efficient with larger sets of data but the simulation factors here are many. Factors such as redundancy, types of transactions, and storage space can heavily skew results.

E. Unresolved Issues and Future Research Topics

In most approaches to concurrency control, the read-set and the write-set are needed to make decisions. These statistics are used in *Optimistic* methods during the validation phase and *Lock-based* methods use them when deciding to acquire locks. This decision making process has been result of many years of research and improving it can lead to less abortions and overall improved performance. A research paper by Sheikhan, Rohani and Ahmadleui [21] describes using an adaptive resonance theory neural-network concurrency control method (NCC) for this decision making process [21]. Through this research they found that NCC increases the degree of concurrency and decreases the number of aborts made by the concurrency controller. This model defines a health factor which is used in comparing different transactions and figuring out which transaction should perform an action. When transactions perform actions, it records the number of aborts they cause with correspondence to the entities they access. It uses this history of transactions to form patterns to build a model around. A comparative analysis of NCC and *Strict Phase Locking* is given in figure 3. The conclusion of this research proved that using this approach gave significant gains over just using *Two-Phase Locking*.

III. Concurrency in Distributed Database Systems

For various reasons, including increased reliability, accessibility, and performance, more applications are relying on a *distributed database*: a single logical database which is distributed across machines and connected over a network [7]. This type of database system presents new and unique concurrency control challenges in order to make the database appear to the user as a centralized database. While doing this, it must also preserve the desirable *ACID* properties while at the same time preserving the benefits of its distributed nature.

Concurrency control techniques for distributed databases must consider the same factors as centralized databases, plus

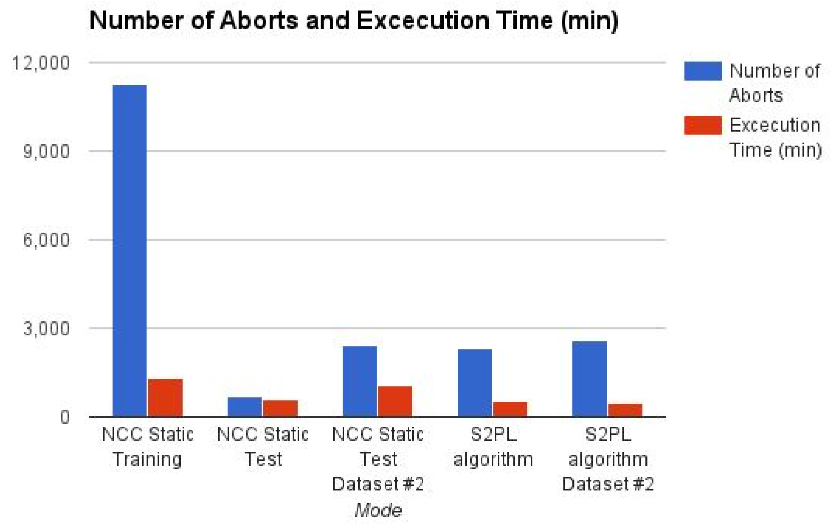

Fig. 3: Comparative analysis of NCC and S2PL[21]

two additional factors specific to distributed databases: *fragmentation* and *replication*.

For performance reasons and to help achieve other desirable properties, distributed databases are divided into logical units called *fragments*, classified as one of *horizontal fragments*, *vertical fragments*, or a hybrid of both. A horizontal fragment is a subset of rows in a table, and a vertical fragment is a subset of columns. Hybrid fragmentation consists of a mixture of both horizontal and vertical fragments [7].

To achieve fault tolerance, fragments are typically copied and maintained at two or more sites. The process of deciding which fragments to copy to which site is called *replication* [7].

Before discussing concurrency control techniques for distributed databases, we must describe how a distributed transaction differs from a centralized transaction. A distributed transaction is typically executed in multiple processes, on multiple machines. A transaction is submitted at a site *S*, then it is broken into smaller sub-transactions to be executed at remote sites. *S* submits these sub-transactions to transaction managers (TMs) at remote sites, and coordinates their activity [20].

We will discuss three concurrency techniques which are designed to account for the fragmentation and replication behaviour of a distributed database: *Distributed Two-Phase Locking*, and *Distributed Optimistic*.

A. Distributed Two-Phase Locking (2PL)

This protocol behaves the same way as the two-phase locking protocol described previously: each transaction first enters a "growing" phase where it may obtain locks but not release any locks, then later enters a "shrinking" phase where it may release locks, but not obtain any new locks [7]. However, due to the fact that the database is distributed across multiple sites, there is extra complexity involved to ensure that all sites have a correct view on which resources are locked at a given time, and to account for scenarios when a site goes offline.

A basic way to implement 2PL in a distributed database is to implement a 2PL *scheduler* at each site: the scheduler would be responsible for receiving lock requests and processing locks according to the 2PL specification. When a request arrives to read/write a resource at a site, the site's scheduler would grant the required lock if it is available, otherwise place the request on a queue to be granted later, once previous locks have been released. This technique is able to handle data replicated across multiple sites for both reads and writes. For a data item *X*, a transaction may read any copy of *X*, so it only needs to obtain a lock on the copy of *X* that it actually reads. To write *X*, a transaction must write to all copies of *X*, so it must obtain a lock for all sites where *X* exists [9].

Other, more advanced distributed 2PL techniques exist such as *primary copy 2PL*, *voting 2PL*, and *centralized 2PL*, about which we won't go into detail.

B. Distributed Optimistic (OPT)

The distributed optimistic algorithm assigns a read timestamp and a write timestamp to every data item. However, transactions can read and write data items freely, storing updates in a local cache until it's time to commit. Each time the transaction reads, it saves the write timestamp associated with the data item. Once the transaction has completed its work, it is assigned a timestamp, which is sent to other sites in a "prepare to commit" phase. A read request is "certified" (in other words, accepted) if:

1. The version that was read is the same as the current version.
2. No write with a newer timestamp has been certified.

A write request is certified if no later reads have been certified [7].

Aside: Apache Cassandra NoSQL Database with respect to Concurrency Control

Apache Cassandra is an open-source implementation of a NoSQL distributed database which is built to handle a large amount of variably structured data. It can be configured to store data across multiple data centers both locally and using the cloud. It offers many benefits such as being able to scale linearly, and no single-point-of-failure. With respect to concurrency control, it does not use common techniques for concurrency such as lock-based or optimistic approaches but instead allows developers to define their own. Consistency for Cassandra is very different than what we have traditionally in other DBMSs. SERIAL reads are one type of transaction Cassandra supports which utilizes uncommitted transactions in validating write sets [24]. The distributed nature of Cassandra leads it to having problems when trying to adopt existing concurrency models. As Cassandra bases itself on a peer-to-peer strategy, no master or centralized database exists. Without a master database it has been analysed by the community and to implement an optimistic-like concurrency control method in Cassandra would prove to be very difficult. An example project which hopes to achieve a traditional synchronization method is Cages [25].

IV. Concurrency with long-lived transactions

There are certain database use cases which involved *long-lived transactions* (LLTs): transactions which take a long time to complete. For example, a transaction which collects statistics across an entire database, or a transaction which waits for input before continuing, say from a user or from a network resource. In general, these are transactions which involve reactive (endless), open-ended, or collaborative (interactive) activities [19]. These types of transactions can be problematic for concurrency: a database using a typical locking technique like 2PL will lock the objects accessed by an LLT until it completes, which may take time to the order of hours or days. During this time, other, shorter transactions will be blocked, leading to poor performance. LLTs also negatively affect the transaction abort rate: LLTs naturally access more data items than other transactions, which increases the frequency of deadlock, thus increasing the frequency of abortions [11].

There is no general way to address the problems of LLTs while simultaneously preserving *ACID* properties; long locking delays and high abort rates will remain no matter what mechanism is used to ensure atomicity [11].

For certain applications, however, it is acceptable to drop the requirement of atomicity. In other words, it is acceptable for an LLT to commit updates and release locks on resources midway through its execution, thus allowing other transactions to acquire these resources and proceed concurrently [11].

[11] introduces the concept of a *saga*: a transaction which is composed of multiple sub-transactions which can be interleaved with other transactions on the system. Each sub-transaction preserves *ACID* properties, but the saga as a whole is not atomic. In most cases, partial executions are still undesirable, so the concept of "compensating transactions" is introduced: each sub-transaction T_i must have a compensating transaction C_i which is to be executed if T_i fails.

In general, a saga is composed of sub-transactions T_1, T_2, ..., T_n and compensating transactions C_1, C_2, ..., C_n. Thus, two execution scenarios are possible:

1. T_1, T_2, ..., T_n
2. T_1, T_2, ..., T_j, C_j, ..., C_2, C_1

In the first, ideal, scenario, all sub-transactions complete successfully. In the second, transaction *j* fails, so compensating transactions *j* through *1* must be executed. The ordering of sub-transactions is not guaranteed and sub-transactions must be independent of one another.

Take the following example database: a table containing car rental reservations, another containing hotel reservations, and another containing flight reservations. The saga *S* wants to book a trip, so it's composed of three sub-transactions: T_1 reserves a car rental, T_2 reserves a hotel room, and T_3 reserves a seat on a flight. Compensating transaction C_1 cancels the car reservation, C_2 cancels the hotel room, and C_3 cancels the flight reservation.

Note that C_i must be semantically opposite of T_i: it's not enough for C_i to simply restore the state of the data item at the time before T_i executed [11]. If T_2 from the above example involved decrementing a count of the available hotel rooms, it is possible for another unrelated transaction *U* to update this count between T_2 and C_2. If C_2 simply restored this count to its previous value, then the effect of *U* is lost, resulting in an inconsistent database. Thus, a valid operation of C_2 would be to instead increment the count, preserving data consistency.

Alternatively, we could treat T_1, T_2, and T_3 as separate transactions, thus guaranteeing ACID and avoiding the need

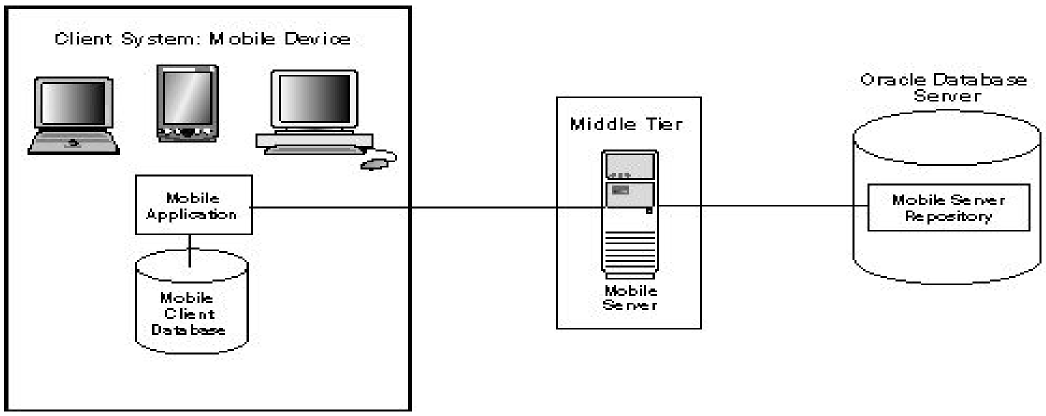

Fig.4 A basic client-server architecture

for compensating transactions. However, the reason for grouping these transactions into a saga comes from an application requirement that a trip booking is atomic; if a car and hotel room are reserved, but the flight reservation fails, we don't want to keep the car & hotel reservations. The decision of whether a sequence of transactions should be grouped into a saga comes from the requirements of the application.

Due to the fact that it does not ensure atomicity of LLTs, it is clear that the saga model has a lower level of correctness than that of 2PL or the other algorithms previously described. There are scenarios when the saga model can lead to unexpected results, such as when a concurrent transaction sees the result of T_i before being compensated by C_i. Given the trip booking example above, imagine that a concurrent transaction R reads the hotels table after T_2 is executed but before C_2 is executed. R may find that all hotel rooms are booked, even though a cancellation is pending. Whereas, using an atomic system like 2PL, T_2 would not have been committed in the first place, so R would have correctly seen that there is an available hotel room.

Sagas are designed to be used in conjunction with an existing transaction protocol that ensures atomicity; sagas are not a substitute for low-level protocols like 2PL. Instead, sagas are implemented on top of such low-level protocols, so naturally, a transaction implemented as a saga will have higher performance overhead than if it was implemented purely in 2PL. This extra overhead consists of the CPU and I/O costs associated with coordinating sub-transactions; e.g. sub-transactions must be individually committed and flushed to disk during the course of the LLT, instead of one large commit at the end.

At its core, the saga paradigm is quite simple. However, it is when considering fault tolerance that the saga paradigm becomes more complex. Algorithms like 2PL rely on logging transactions' operations to a file, and undoing actions in this log in the event of a system crash. With sagas, however, it is not enough to simply log the actions that occurred; we must also account for pending compensating transactions: these must be executed during system recovery to restore the database to a consistent state. Therefore the DBMS must have a durable mechanism to know about which future compensating transactions must be executed and how to execute them. One straightforward way to do this is to store the saga's code in the database itself at the beginning of the saga, as T_1. If the system crashes during the saga's execution, then a recovery mechanism can read the saga's code and continue where it left off [11]. While the concept of sagas is quite simple, the fact that sagas must be used alongside an existing transaction algorithm, and handling edge cases like system failures, cause it to have a high complexity.

It is worth mentioning that the saga paradigm is not specific to centralized database management systems; it is also applicable to distributed database management systems [11] and applications which adopt a *service-oriented architecture* (SOA) [18].

A topic for future research is discussed in [19]: *ACTA* is a framework which attempts to generalize transaction models around four factors: visibility, consistency, recovery, and performance. In [19], ACTA is used to formally characterize the saga model by proving properties of sagas and applying variations to the saga model based on this characterization. By utilizing the ACTA framework, it may be possible to characterize other transaction models as well, such as those that operate on mobile databases, as we will discuss next.

V. Concurrency in Mobile Databases

Mobile databases have become increasingly relevant with the exponential growth of wireless technology and processing power. Stock trading systems, security networks, and directional map systems on mobile devices are all such manifestations of this emerging phenomena.

A *mobile database* is a database that exists on a mobile device. We look at mobile databases in the frame of a client-server architecture shown in figure 4. The nature of this architecture is *distributed*.

Mobile databases introduce several additional constraints to the task of concurrency control in databases. A mobile database needs to account for the following factors:

➢ Restricted bandwidth

➢ Limited battery power, memory and processing power

➢ Communication delay

➢ Disconnections

➢ Users with multiple devices need to *replicate* their data across devices

For a general DBMS, we can split concurrency control methods into pessimistic and optimistic control methods. They also apply to mobile devices and have their own set of disadvantages.

Pessimistic concurrency control methods are the set of concurrency control methods that use some form of data lock to maintain data integrity. Pessimistic techniques show poor performance mainly due to unacceptably long periods of locking in the case of weak connection or even disconnection. [15].

Optimistic concurrency control methods are the set of methods that validate the transaction while running and abort if there is a conflict. The usage of optimistic control schemes works better, but updates need to be propagated and validated by the central database, Since mobile transactions are *long-lived*, standard optimistic control schemes result in a high number of conflicts [13].

Due to potential inconsistencies in communication, standard concurrency control methods do not generalize to mobile databases and raise issues of deadlock, starvation, data inconsistency, and high abortion rates [15]. This paper discusses several mobile transaction processing models which increase concurrency in mobile database systems.

Dynamic Clustering

[13] introduces *dynamic clustering* a flexible, two-level consistency model to address disconnection in mobile databases and maintain database consistency.

The central database is merged with connected mobile databases within its cluster during strong connection. When a mobile database is disconnected it creates its own cluster, which can differ from the central cluster. Every data object within a database has two versions:

- *Strict Version: A* globally consistent version operated on by standard read and writes which are only executed when mobile databases are connected to the central database.
- *Weak Version*, which is locally consistent in the cluster but not necessarily globally consistent. Weak transactions take place when mobile databases are weakly connected or disconnected.

If a mobile database makes changes while offline, it makes a local commit with weak transactions, otherwise if it is connected it will use strict transactions. Weak transactions release their locks at local commits, and strict transactions release at global commit.

When a weakly or disconnected mobile database reconnects, the database synchronizes it to a consistent state. During synchronization, locally committed transactions may be rolled back as a result of conflicts. In this case, a function *h* is defined that can limit the number of local commits, and number of inconsistent copies. Depending on specifications, the function *h* uses four conflict tables to reflect strict operations on weak versions [17].

Two-Tier Replication

In a two-tier replication scheme, each object has a set of copies and a master copy. It supports two types of transactions, *base transactions* and *tentative transactions.* Base transactions operate on the master version and tentative transactions operate on the copies. Tentative transactions take place on the mobile database and will update the local data on the mobile database only. Upon connection to the database, selected tentative transactions will be become base transactions so that there is global consistency. If there are conflicts, there is a relative “acceptance criterion” in place to make decisions; this is similar to clustering in that locally committed transactions, in this case tentative transactions, can be rolled back. Upon connection to the database, selected tentative transactions will become base transactions so that there is global consistency [17].

Prewrites

[17] addresses data availability on mobile databases by introducing *prewrites*. Prewrites are future data states that are subsets of the actual *write value,* meaning they have the property of small memory footprints on the mobile database.

Once the prewrites have been processed, the mobile transaction pre-commits to the mobile database. Precommitted results are visible to other hosts before commit to minimize blocking. A precommitted transaction is guaranteed to commit, which avoids the compensating transactions and rollbacks that clustering/2-tier transaction models have which can be costly. A pre-read always returns a prewrite, but a write returns a read.

VI. Conclusion/Future Work

Due to the increased need for databases to support millions of transactions, concurrency is needed for acceptable response times. This need brings alongside many problems which are formalized in the community [4] and presented in this report such as *The Dirty Read Problem* and *The Non-Repeatable Read Problem*. Different solutions exist to combat these problems and their categorizations are presented in [8]. Analyzed is also the Oracle Database Management System [6] and how it incorporates different isolation levels for different use-cases. All of the solutions have different advantages and disadvantages which make them a popular topic of study. Our classification scheme was used to critically assess the different methods using data from [8]. An experimental method [21] which looks at introducing a neural-net-based transaction decision-making system which proved itself to be more efficient than 2PL. Looking at current trends, distributed database systems were discussed in depth using evidence from [7] and [9]. Looking deeper, we found sagas to be viable method to handle long-lived transactions. Analyzed in [11], we see how it integrates into a concurrent, distributed environment. A future look into *ACTA* was briefly made to give a formalized look at the saga paradigm. Another trend was looked at was mobile databases. Research [15] in this field is rapidly advancing because of the need for distributed databases residing on mobile devices. Different transaction models pertaining to mobile databases were discussed.

References

[1] Meraji, Sina, “Concurrency Control”, Slides pp. 1-45, Nov 15 2016.

[2] Oracle, "Database Concepts", Chapter 10 Transactions, pp. 17. Available: https://docs.oracle.com/database/121/CNCPT/transact.htm#CNCPT016

[3] Delaney, Kalen and Guerrero, Fernando, "Database Concurrency and Row Level Versioning in SQL Server 2005", Technet, April, 2005, Available: https://technet.microsoft.com/en-us/library/cc917674.aspx

[4] Microsoft/Community, "Concurrency Effects, pp. 1 Available: https://technet.microsoft.com/en-us/library/ms190805(v=sql.105).aspx

[5] Kuo, Tei-Wei, Ming-Chung Liang, and LihChyun Shu. "Abort-Oriented Concurrency Control for Real-Time Databases." *Computers, IEEE Transactions on* 50.7 (2001): 660-73. Web. 26 Nov. 2016

[6] Oracle. "Oracle® Database Concepts", 9 Data Concurrency and Consistency. Available: http://docs.oracle.com/database/121/CNCPT/consist.htm#CNCPT1319

[7] Kaur, Mandeep and Kaur, Harpreet, “Concurrency Control in Distributed Database System.” International Journal of Advanced Research in Computer Science and Software Engineering, Volume 3, Issue 7, July 2013.

[8] Bhargava, Bharat, "Concurrency Control in Database Systems". IEEE Transaction on Knowledge and Data Engineering, Vol. 11. No. 1, January/February, 1999

[9] Bernstein, Philip and Goodman, Nathan. “Concurrency Control in Distributed Database Systems”. Computer Corporation of America, Cambridge, Massachusetts 02139.

[10] Figure 4: tutorialspoint.com, "wait_for_graph.png". Available at: https://www.tutorialspoint.com/dbms/images/wait_for_graph.png

[11] Garcia-Molina, Hector and Salem, Kenneth, “SAGAS”. Department of Computer Science, Princeton University, 1987

[12] Serrano-Alvarado, Patricia, Claudia Roncancio, and Michel Adiba. "A Survey of Mobile Transactions." Distributed and Parallel Databases 16.2 (2004): 193-230. Web.

[13] Pitoura, E., and B. Bhargava. "Maintaining Consistency of Data in Mobile Distributed Environments." Proceedings of 15th International Conference on Distributed Computing Systems (n.d.): n. pag. Web.

[14] Bakura, Sirajo Abdullahi, and Aminu Mohammed. "Lock-free hybrid concurrency control strategy for mobile environment." *2014 IEEE 6th International Conference on Adaptive Science & Technolo*

[15] Moiz, Salman Abdul, Lakshmi Rajamani, and Supriya N. Pal. "Commit Protocols in Mobile Environments: Design & Implementation." *International Journal of Database Management Systems (IJDMS)* 2.3 (2010).

[16] Madria, Sanjay Kumar, and Bharat Bhargava. "A transaction model to improve data availability in mobile computing." *Distributed and Parallel Databases* 10.2 (2001): 127-160.

[17] Serrano-Alvarado, Patricia, Claudia Roncancio, and Michel E. Adiba. "Mobile Transaction Supports for DBMS." BDA. 2001.

[18] McCaffrey, Caitie, “Applying the Saga Pattern.” *GOTO Chicago 2015 Conference*. Available: http://gotocon.com/chicago-2015/presentation/Applying%20the%20Saga%20Pattern, https://www.youtube.com/watch?v=xDuwrtwYHu8

[19] Chrysanthis, Panos K. and Ramamritham, Krithi, “ACTA: The SAGA Continues.” *Database Transaction Models For Advanced Applications*, 1992

[20] Ramakrishnan, Raghu and Gehrke, Johannes, “Database Management Systems, Third Edition.” 2003

[21] Sheikhan, Mansour and Rohani, Mohsen and Ahmadluei, Saeed, "A neural-based concurrency control algorithm for database systems". July 21, 2011

[22] Butrico, Maria, et al. "Enterprise data access from mobile computers: an end-to-end story." Research Issues in Data Engineering, 2000. RIDE 2000. Proceedings. Tenth International Workshop on. IEEE, 2000. APA

[23] Figure 4: :docs.oracle.com, "genarch.gif". Available at: https://docs.oracle.com/cd/E22663_01/doc.11100/e22677/genarch.gif

[24] Datastax, "About transactions and concurrency control", Available at: https://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_about_transactions_c.html

[25] Williams, Dominic, "Locking and transactions over Cassandra using Cages", Available at: https://ria101.wordpress.com/2010/05/12/locking-and-transactions-over-cassandra-using-cages/