Scalable Database Management System (DBMS) Architecture with Innesto

by

Primal Wijesekera

B.Sc, University of Colombo, Sri Lanka, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

October 2012

Abstract

Database Management systems (DBMS) have been in the core of Information Systems for decades and their importance is getting higher and higher with current high growth in user demand and rising necessity to handle big data. With recent emergence of new style of deployments in the cloud, decades old architectures in DBMS have been greatly challenged due to their inability to scale beyond single computing node and to handle big data. This new requirement has spawned new directions along scaling data storage architectures. Most of the work surfaced lacks the applicability across many domains as they were targeting only a specific domain.

We present a novel scalable architecture which is implemented using a distributed spatial partitioning tree (SPT). This new architecture replaces only the storage layer of a conventional DBMS thus leaving its applicability across domains intact and provides strict consistency and isolation. Indexing and locking are two important components of a Relational Database Management System (DBMS) which pose as potential bottleneck when scaling. Our new approach based on SPT provides a novel scalable alternative for these components.

Our evaluations using the TPC-C workload show they are capable of scaling beyond single computing node and support more concurrent users compared to a single node conventional system. We believe our contributions to be an important first step towards the goal of a scalable, cloud aware and full-featured DBMS as a service.

Preface

All the work described in this thesis was performed under the supervision of Andrew Warfield and with regular consultation from Norm Hutchinson. The entire project described in the thesis is implemented as a teamed up work with Mahdi Tayarani Najaran, a current PhD student in the NSS lab. Ken Salem from University of Waterloo helped a lot in guiding the project in the early phase.

The Innesto project (Section 2.2) described in the thesis is entirely developed by Mahdi as part of his PhD thesis work. Since the Indexing sub system (Section 4.1) fully used the Innesto as it is, it also belongs to his work. Distributed locking described in (Section 4.1) needed for correct isolation semantics, is collectively designed and implemented with Mahdi. In the Transaction Handler (Section 4.3.2), authors' main contribution is in the Data operations group where core CRUD operations are implemented. The storage handler (Section 4.3.4) responsible of communication between the QPU and the Transaction Handler and the correct handling of MySQL instructions is a work of the author.

The project described in Appendix C which was the first step in this project was also a work of the author. Work described in Appendix B has been published separately as Distributed Indexing and Locking: In Search of Scalable Consistency, Mahdi Tayarani Najaran, Primal Wijesekera, Andrew Warfield, Norman C. Hutchinson, 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS 2011).

Table of Contents

Abstract . . . . **ii**

Preface . . . . **iii**

Table of Contents . . . . **iv**

List of Figures . . . . **vii**

Acknowledgements . . . . **viii**

1 Introduction **1**

2 Background **4**

2.1 Current RDBMS Architecture 4

2.1.1 Query Processing 4

2.1.2 Indexing 5

2.1.3 Locking 6

2.1.4 Known Bottlenecks 7

2.2 Innesto 8

2.2.1 Spatial Partitioning Tree 9

2.2.2 Two Phase Commit 10

3 Design **11**

3.1 Design Objectives 11

3.1.1 General Applicability 11

3.1.2 Scalability without Higher Level Partitioning 11

3.1.3 Provide Strict Isolation and Consistency 12
3.1.4 Build a Platform for a DBaaS 12
3.2 High Level Design 12
3.3 Design Challenges 14
3.3.1 Distributed Indexing 14
3.3.2 Distributed Locking 14

4 Implementation 15
4.1 Distributed Indexing 15
4.1.1 SPT for Indexing 15
4.2 Distributed Locking 17
4.2.1 Logical Locking with SPT 17
4.2.2 Range Based SPT 18
4.2.3 Locking Semantics 18
4.2.4 Fair Queueing vs FIFO 18
4.2.5 Deadlock Detection 19
4.2.6 Workload Profiling 19
4.3 Higher Level Architecture 20
4.3.1 Index and Lock Manager, Memnodes 21
4.3.2 Transaction Handler 22
4.3.3 Query Processing unit 25
4.3.4 Storage Handler 26

5 Evaluation 27
5.1 Setup and Benchmark 27
5.2 Strong Scaling 28
5.3 Contention 29
5.4 Web Compatible Workloads 30
5.5 Range Queries 32
5.6 Crash Consistency Cost 33

6 Related Work 36
6.1 Data Partitioning 36
6.2 Key-Value Stores 37

6.3 Scaling Storage 37
6.4 Scalable Transactional Models 38
6.5 Distributed Indexing 38
6.6 Distributed Locking 39
6.7 Commercial Offerings 39

7 Conclusion and Future Work 40
7.1 Conclusion 40
7.2 Future Work 41

Bibliography 43

A InnoDb Configuration 47

B SinfoniaEx 48

C MySQL over Cassandra 51

List of Figures

Figure 2.1 The Conventional Workflow in a DBMS 5
Figure 2.2 High Level Innesto Architecture 8
Figure 2.3 Three Dimensional Spatial Partitioning Tree 9

Figure 3.1 Scaling the Storage Layer 13
Figure 3.2 Scalability with Query Processing 13

Figure 4.1 Spatial Partition Tree as DBMS Index 16
Figure 4.2 Adding a Primary Key in SQL 16
Figure 4.3 Querying Index in SQL 17
Figure 4.4 Full System Architecture 21
Figure 4.5 Current MySQL Architecture 25

Figure 5.1 Strong Scaling in Query Processing 29
Figure 5.2 Scaling with Users 31
Figure 5.3 User Contention 32
Figure 5.4 Web Compatible Workload 33
Figure 5.5 Range Queries 34
Figure 5.6 Cost of Crash Consistency 35

Figure B.1 SinExTree Thrashing with High Contention 49
Figure B.2 SinExTree as an Index 50

Acknowledgements

I owe a debt of gratitude to God for all the blessings showered upon me and especially giving me the opportunity to work in a nice place like NSS lab. This work would not be possible without the support I got from Andrew Warfield as my supervisor. After allowing me to work on my own research area; ideas, guidance, insightful discussions and encouragement I received from Andy assisted tremendously to shape the thesis in a better way. I also would like to thank Norm Hutchinson for numerous thoughtful discussions we had and for the valuable comments as the second reader to this thesis. I would also like thank Kenneth Salem for the help given to start the project along the correct direction and useful feedback given at numerous times throughout the project.

The work presented in the thesis is the result of an collective effort put together with Mahdi Tayarani. His help throughout the course of the project has helped immensely to reach the state where it is now. I would also like to thank Charles ’Buck’ Krasic for shaping my research in the early stage and for supervising me during the MITACS intern-ship. Alan Wagner provided useful guidance as my advisor when I first came into UBC.

Several friends and colleagues too many to mention individually have offered their help in numerous ways from sharing ideas to providing good company and cheer.

Finally I am particularly grateful to my family for their immense support. I am here today due to the sacrifices made by and continue to make by my family. Especially for providing the necessary encouragement during ups and downs throughout the course of the program.

Dedicated to my Parents and Sister,
for all their support.

Chapter 1

Introduction

Relational Database Management Systems (RDBMSs) have been an integral part of computing environments for decades and will likely to play that role for the foreseeable future. This is mainly due to its generality, simplicity and importantly due its expressibility in terms of querying. With the emergence of cloud computing and hosted services as an attractive new model of systems deployment, Database-as-a-Service (DBaaS) surfaced as an effective mechanism to provide the luxury of a DBMS in the form of a service hosted in a cloud environment [1, 2, 5]. It has its advantages mainly in the perspectives of administration (most of it are taken care of by the cloud provider), in terms of cost(pay-per-use) and resource provision based on demand (scale in and scale out).

However, majority of the database service offerings have not being able to provide one of the most crucial features of hosted services to its clients, i.e. scaling beyond one computing node. Amazon RDS [1] provide the feature of scaling out into a bigger machine when the demand goes high. However the maximum it can go is limited by the most powerful computing node it has to offer[38]. Scalability of DBMS has been in the center of attention due to growing necessity for scaling (both in terms of query processing power and storage) in response to steep growth in user demand for applications across the spectrum.

Data partitioning has been the foremost approach to solve the scalability issue and the work still continues to find ways of efficiently partitioning data to reduce costly distributed transactions [19, 32]. However with current complex data querying requirements its often hard to come up with a good partition scheme that both helps to scale and reduce the number of distributed transactions. On the other hand, there has been some work directed towards lesser consistency levels than strict ACID compliance consistency and lesser expressibility than SQL [17, 22]. Although such models provide greater scalability, it has been found [14] that these new abstract and less consistent data models give hard time to developers to develop stable, error-prone systems as most of the heavy work done in DBMS are pushed to the application level making the application even more complex. Most of those systems are tailored to serve specific type of workloads losing its generality.

The thesis describes an architecture based on iEngine (Innesto-Engine), a memory-resident distributed database storage engine. The described architecture scales without requiring to partition the data and worrying about distributed transactions touching multiple partitions. It does not compromise strict consistency, atomicity or isolation for greater scalability thus provides the same luxuries of a conventional DBMS. It also offers options for crash-recoverability and durability.

Few of unique features of the system compared to the rest of available alternatives are given below.

- **Scalability**: iEngine can scale adding either more query processing power or more memory storage or both. This is done without partitioning the data space at all and with strict consistency(Serializability) intact.

- **Workload Generality**: iEngine scales both in the presence of simple web based (lookup and inserts) operations and in the presence of complex range operations.

- **Graceful Degradation**: Novel techniques used in locking enables iEngine to scale well across many nodes and handle high contention quite well with out degrading steeply.

- **Architectural Support**: iEngine is a general purpose storage engine which can work with any modern RDBMS engine and architecturally supports being used by different engines simultaneously.

The iEngine is based on Innesto which is a distributed in-memory Spatial Partitioning Tree (SPT). Innesto functions similar to Sinfonia [12] and shares many techniques with it. A distributed SPT provides an efficient alternative for conventional BTree based indexing done in RDBMS. iEngine avoids many of the bottlenecks found in traditional storage engines [23]. Although logging is not mandatory in iEngine and it can provide recoverability if the user needs it. All the components of the system are single threaded thus avoids the overhead of latching. Buffer management is not necessary since everything is stored in memory. Although iEngine follows the conventional two phase locking, it can scale into multiple nodes when needed thus locking does not become a bottleneck.

Chapter 2

Background

2.1 Current RDBMS Architecture

2.1.1 Query Processing

As it is shown in Fig 2.1, components in a conventional DBMS architecture can be categorized into SQL processing and storage. When a query first comes in, it will be checked for correctness of SQL semantics and planned for optimum execution strategy by the SQL processing components. In deciding the execution plan, factors like number of rows to be fetched from storage, whether to use index or to use a table scan, which key would fetch less number of rows are taken into consideration in order to return the data as quickly as possible to the user. Meta values such as size of index, average scan time, index lookup, delete times for a given table are usually fed into the above process by the underlying storage component in order to come up with an efficient execution plan. It is this component that provides the rich querying feature in the DBMS.

It was the common understanding that the query processing stage is a bottleneck in scaling the current architecture beyond a single machine. This understanding spawned a new breed of data storage systems mainly focusing on the systems' aspect of the architecture without the support of SQL (See Section 6.3) . However this understanding later proved to be wrong to greater extent with findings of bottlenecks in the storage layer [23]. There is some work done on scaling query

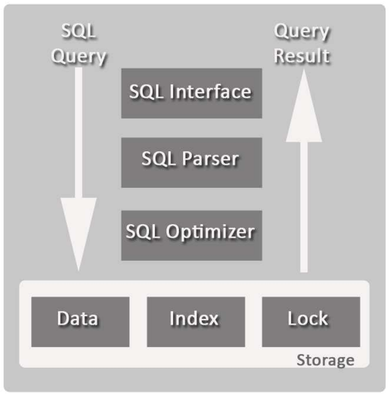

Figure 2.1: The Conventional Workflow in a DBMS

processing itself as well. Implementing SQL Engines using map-reduce [30] is one such example of an attempt to scale SQL Processing by paralleling the sequential SQL work flow.

2.1.2 Indexing

Once an execution plan is decided, data is fetched from the storage layer. Data usually resides in the memory or in the hard disk. When hardware is provisioned Database Administrators (DBA) make sure that the entire data set (or at least the index) will reside in the memory for most of the time to reduce the overhead of disk seeking. Storage layer could be serving expensive tables scans or giving data from the index. Some requests can be entirely served from the index without reaching out for the data (e.g. coverup index) If the query cache is configured, some queries can be served by the query processing unit itself without waiting for the storage.

In serving index requests, some queries are index lookups with a direct map to a key stored in the index or index range query which will most likely scan and return a sub-tree or a slice of an index. Functionality of range queries entirely depend on the type of the index being used. A few of the options for indexes are hash based indexing, BTree based indexing, and RTree based indexing. While hash based indexing provides the fastest mechanism to access the data it can not support range queries efficiently. BTree based approach is the most commonly used method and supports both lookups and range queries. The RTree based approach is mainly used to store Spatial information for Geo information systems.

Whether to use an index lookup or a range query is decided by the query processing unit based on the available keys. Storage layer can force the query processing unit to always use an index without going for an expensive table scan by providing meta data accordingly. Thus defining an index for each SQL table makes execution a lot faster.

In scaling current DBMS architectures beyond a single machine, indexing plays a key role as there are not any accepted mechanisms to provide distributed indexing with all the features provided in the current mechanism. While hash based indexing is used in some systems, its inability to support range queries efficiently greatly affects the querying capability thus it usage is only limited to few use cases.

2.1.3 Locking

Locking is the sole component responsible of ensuring that the correct isolation level is enforced among concurrent transactions by locking data items / data ranges touched by each transaction. However locking only happens if the current query is in the middle of a transaction which has correct isolation level defined. The majority of DBMSs support 4 levels of the isolation levels as given below in from lowest level to most highest isolation level.

- **Read Uncommitted**: There is no isolation among transactions. Any transaction can see intermediate results of other transactions.

- **Read Committed**: Transactions are allowed to see values only committed by other transactions. Transaction can see values committed even after the beginning of current transaction.

- **Repeatable Reads**: Transactions can see only values committed before the beginning of the current transaction.

- **Serializable**: Even values not there (but could have been accessed if its were there) are locked to avoid phantom problems. (Eg. Select * from index <70; would lock the entire data region below 70 avoiding any new value addition until the given transaction is terminated.)

Each isolation level differs from the others in deciding when to release the locks they have acquired in the current context. To support these different isolation levels, it has to support both item locking and complex range locking. Most common approach to implement locking is to use MVCC (Multi Version Concurrency Control) which enforces the isolation by not locking but by keeping different versions as seen by each transaction but its vulnerable to high contention.

Distributed locking has been implemented in some systems (see Section 6.6) however little has been done in the context of DBMS locking. The requirement to support both range locks and support the scalability is not trivial in a distributed setup.

2.1.4 Known Bottlenecks

Apart from scalability issues there are couple of known bottlenecks even in the single node setup that hinders high performance [23]. And there are lots of systems that try to resolve at least one or two of these bottlenecks to get higher performance. The proposed system also addresses many of the below mentioned issues.

- **Logging**: Logging data for recovery happens in several layers in the DBMS and tracking changes to log incur lots of overhead too. Solution would be to achieve recoverability by other means incurring low overheard compared to logging. This also generates excessive amounts of I/O requests which is another overhead.

- **Locking**: Traditional two phase locking is usually managed by a centralized separate entity could also be a source of overhead.

- **Latching**: Modern DBMS are multi-threaded to fully use the available cores in modern systems however many shared data structures have to latched before using it.

- **Buffer Management**: This creates an unnecessary indirection on each record access.

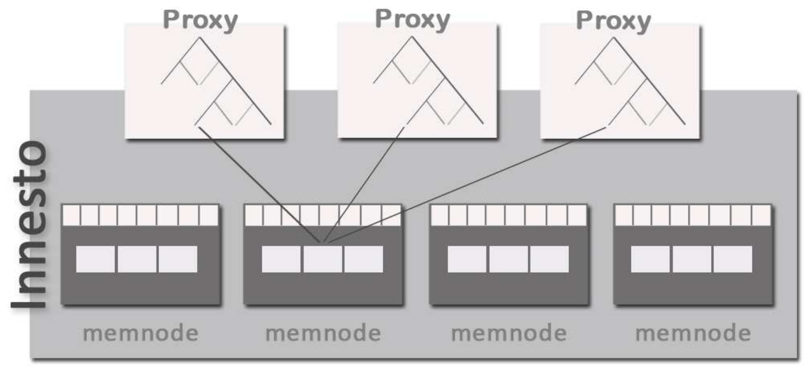

Figure 2.2: High Level Innesto Architecture

2.2 Innesto

Innesto is a scalable key/value storage system based on Distributed Spatial Partitioning Tree (SPT). As shown in Figure 2.2, Innesto has a two layered architecture, where the proxy is a stateless component and the memnode store everything. When a request comes in, the proxy does the tree traversing which is cached and communicates with memnodes to get the data in leaf nodes.

Innesto is inspired by Sinfonia [12], which emphasized that developing fault tolerant distributed systems can be done with less hassle with distributed data structures than using complex communication protocols. Memnodes make sure that the data stored in is it consistent and locks the particular data for the time of a given operation avoiding others to modify it. Locking is quite similar to traditional read / write locks found in DBMS. Innesto has the notion of mini-transactions which are similar to mini-transactions proposed in Sinfonia. While Sinfonia has non-blocking versions, Innesto has the capability to block a mini-transaction until it acquires all the locks needed. Together with mini-transactions and light weight two-phase commit, Innesto provides atomicity for each operation executed on the SPT.

Each of these components are single threaded and communicate via a custom built stream layer. Memnodes store all the data in the memory thus no disk seeks

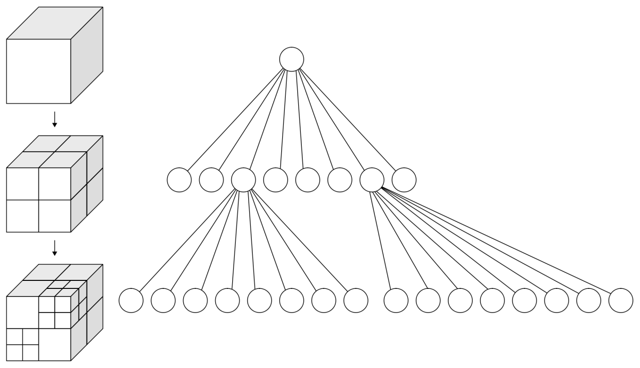

Figure 2.3: Three Dimensional Spatial Partitioning Tree

are required to serve the data. However, if a requirement arises a memnode can synchronously log the data to the disk to minimize the data loss in an event of a crash or hardware failure.

2.2.1 Spatial Partitioning Tree

Spatial partitioning tree is a hierarchical data structure organized into a tree, which splits the same data region recursively when a particular region gets congested. In Innesto as the regions get split, these new regions (referred to as node partitions from here on) will be equally distributed across available memnodes to balance the load. This notion of node partitions technically enables Innesto to migrate live partitions across nodes giving scaling out and scaling in features. Figure 2.3 from [8] shows how a three dimensional spatial partitioning tree splits its self from a one partition to multiple partitions.

Due to the hierarchical nature of SPT; its a better candidate for insert intensive workloads compared BTree based approach. However SPT is not a balanced tree. In a multi-dimensional SPT, serving multi-attribute range queries are implemented more efficiently than in a BTree. It enables more fine grained range queries compared to a BTree.

2.2.2 Two Phase Commit

To achieve atomicity in all operations performed on the SPT, a light weight two phase commit is implemented. Each operation on SPT is executed using a mini-transaction. The proxy that initiates the operation will act as the coordinator. When multiple memnodes are involved in a given mini-transaction, each memnode will try to lock the data region it tries to access and send a OK vote upon acquiring the lock or FAIL vote if the lock request times out. Coordinator will send back the final commit message to all the memnodes based on the VOTE in the first phase. If there is only one memnode, it will go ahead and commit the changes as soon as its vote is sent to the coordinator making the commit execution faster. If the proxy has a stale information on partitions it will be notified in the first phase and proxy will retry the mini-transaction after updating the stale cache entries.

Chapter 3

Design

3.1 Design Objectives

3.1.1 General Applicability

Most of the products in the market or research carried out requires current DBMS users to do intensive migration such as entirely re-writing the data management layer to comply with new products. Although those products provide great features only handful of current users want to try those due to the cost of moving into the new products.

Our main design objective is to make the migration into our platform as smooth as possible. Yet the system has to provide the scalability we want to achieve. This way users do not have to change their application comply with the new DBMS.

3.1.2 Scalability without Higher Level Partitioning

The de-facto partitioning method for scaling out DBMS was to partition the data based on a particular access pattern E.g., based on geography, based on departments, etc. This reduces the costly cross partition transactions. With the modern complex workload patterns it could not be a trivial task to partition data based on a simple partitioning scheme or to avoid distributed transactions. And after certain point the number of partitions could grow beyond control.

To avoid such hassles our system has to scale without any user intervention such as data partitioning. System has to be able to distribute load across available nodes so that the load is balanced and there should not be hot spots.

3.1.3 Provide Strict Isolation and Consistency

Many systems that require high scalability sacrifice strict consistency and isolation in favour of the scalability requirement. Such systems work perfectly well for selected set of domains with very high scalability requirements. However there are many with the scalability requirement yet unable to sacrifice the consistency. It was also revealed that their lesser consistency and isolation models make the application layer even more complex than before with handling the bulk of the work that were used to be handled by the DBMS.

Our application should provide scalability without compromising the consistency or the isolation usually provided with the DBMS. This will cater to the needs of those who require both scalability and the high consistency or for those who already have a complex application layer and can not afford to make it even more complex.

3.1.4 Build a Platform for a DBaaS

Even though there is a hype about offering DBMS as a service, there are very few offerings that truly provide the DBMS as a service. Many still battle to scale beyond one single machine or to provide both high consistency and scalability at the same time as in a conventional DBMS.

Although this project would directly implement a DBMS as a service, this should serve a first step towards such a goal in providing truly scalable DBMS with guaranteed consistency and isolation. This requirement would also bring, the need to support multi tenancy in the system.

3.2 High Level Design

Our approach to this problem is to design a general purpose, fully transactional and distributed storage engine so that current DBMS could replace the current single node storage with the distributed storage layer. To achieve that, we have to replace

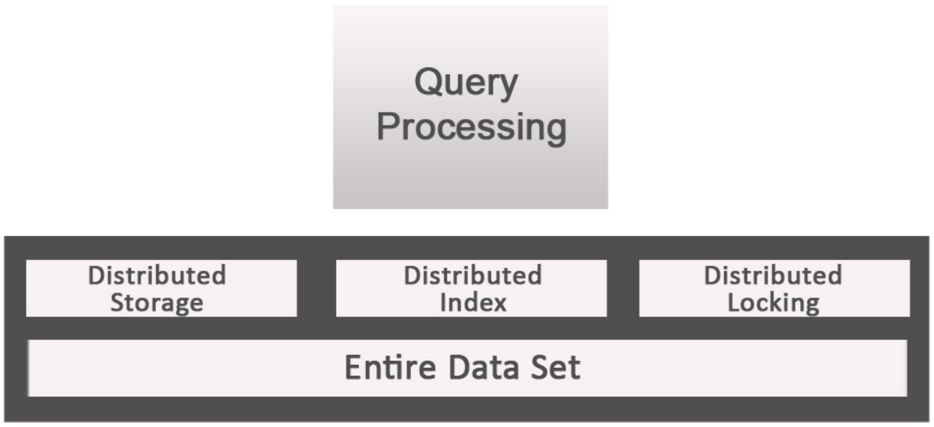

Figure 3.1: Scaling the Storage Layer

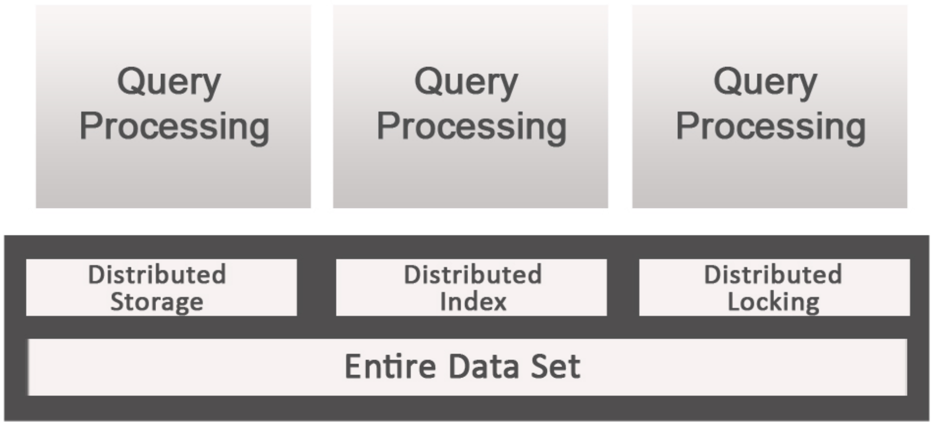

Figure 3.2: Scalability with Query Processing

single node indexing, locking and storage with distributed indexing, distributed locking and distributed storage. Such a layered design is shown in Figure 3.1.

To avoid manual data partitioning, when the system scales out, the indexing and locking should cover the entire data set not only data local to the current machine as happens in data partitioning. Scaling the storage layer would give the system scalability along more storage and adding more query processing units (as shown in Figure 3.2) the system can scale for more processing power for a fixed size data set. Thus this design provides the scalability along two dimensions.

3.3 Design Challenges

The above mentioned design has several key challenges to overcome to make it a practical solution.

3.3.1 Distributed Indexing

One of the main reasons for partitioning being the de-facto for some time, is the need for a good practical solution for distributed indexing. While hash based indexing provides good scalability it lacks the rich querying features needed by DBMS.

3.3.2 Distributed Locking

Distributed locking has not been explored as it is in distributed indexing. Although scalable distributed locking has been already implemented using techniques such as Paxos (See Section 6.6) there are not many practical solutions generic enough to support all the complex locking semantics found in DBMS, especially to support range locks needed to over come the phantom problem.

Chapter 4

Implementation

4.1 Distributed Indexing

The foremost challenge faced during the implementation was to come up with an efficient, practical distributed indexing subsystem. Although the proposed Distributed Btree [13] provides a good alternative as its quite similar to what is found in the single node system. However it lags in performance in insert heavy workloads. While Innesto (See Section 2.2) is being developed in the NSS lab mainly for multi-user games, the distributed SPT used in Innesto has advantages over the BTree based approach. Thus decision was taken to use the Spatial Partition Tree to store standard non-spatial data.

4.1.1 SPT for Indexing

Figure 4.1 shows how a three dimensional SPT is used to index orders table in the TPC-C Benchmark. Orders table has three index attributes and each attribute is represented as a dimension in the SPT. Representing each attribute as a dimension gives finer grained querying compared to BTree.

Figure 4.2 shows the SQL to add the primary key to *orders* table in the order of warehouse id, district id and order id. Since the key will be a concatenation of three parts in the conventional BTree based indexing, each query should have the value for the warehouse id even to query using any other value and to query with order id query should have the value to the first two components. Most of the

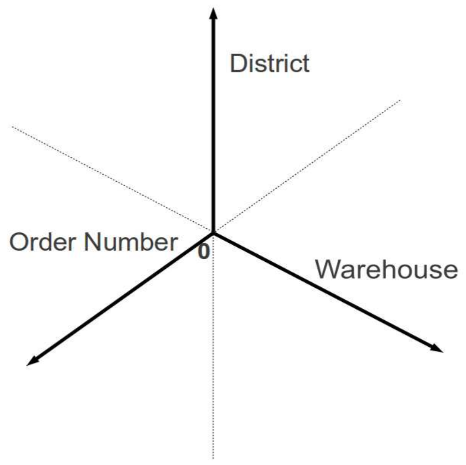

Figure 4.1: Spatial Partition Tree as DBMS Index

```
ALTER TABLE orders ADD CONSTRAINT pkey_orders PRIMARY KEY (o_w_id, o_d_id, o_id);
```

Figure 4.2: Adding a Primary Key in SQL

current DBMS designed to work with BTree based systems, are ignoring efficient index queries and go for costly table scans if the first component is missing.

With a separate dimension representing each part in the key, SPT can serve any type of query without depending on the previous parts in the index. Figure 4.3 shows range query where it requests all the orders that have been made in district 4 across all the warehouses. With SPT storage will see a request like {*,4-4,*} saying for the first and last parts of the request can get any values but the middle part should have the value 4 (with min value = max value represents an equal condition). Support to these types of light weight OLAP type queries is becoming essential as modern systems are being used on both OLTP and OLAP type workloads.

With each region recursively splitting in the case of congestion on regions due to inserts, parent nodes keeps unchanged on splitting. This will make sure only leaf nodes gets affected and the rest of the tree hierarchy remains constant. Although this creates an un-balanced tree, this is very effective in terms of insert intensive

```
Select * from orders where o_d_id = 4;
```

Figure 4.3: Querying Index in SQL

workloads since cached hierarchies in proxies remains valid avoiding unnecessary cache reloading. And irrespective of the logical locking described below, each index operation locks the data it touches for the period of the operation to further guarantee the isolation.

Modern DBMSs have similar options for spatial data indexing using multi dimensional storage however, those options are not available for storing non-spatial data while BTree based indexing is the most common option.

4.2 Distributed Locking

Locking in DBMS has some complex requirements to be fulfilled to be able to serve the rich SQL features. While distributed locking itself is a costly operation supporting distributed range queries is an added complexity. Range locks are required to prevent phantom issue to make sure that even results not returned in a query will remained locked until the given transaction is terminated preventing some other transaction making an insertion. With these two requirements on hand, the decision was to use the SPT for locking as well. As in with the indexing, in the locking each different attribute is represented as a dimension in the SPT.

4.2.1 Logical Locking with SPT

Mandatory locking and logical locking are two options for ensuring isolation among concurrent transactions. While mandatory locking is usually hardware enforced locking such as page level locking, logical locking is a higher level locking where existence of an data item in a data structure is interpreted as the given item is locked by some transaction. With SPT being the underlying data structure, current system uses logical locking. An entry in the tree interpreted as that particular data item or data range is locked by another transaction. Removal of an data item or a range means releasing the lock held by that transaction.

4.2.2 Range Based SPT

The original SPT which is working with data items not with ranges, is changed to work with ranges instead of items in the locking subsystem. The flexible model under which the SPT was developed made the transition easier while requiring only new handlers to deal with ranges. Thus each item in the locking SPT is a range; a single item is represented as a special range where both min and max have the same value. Thus most of the range locks could spread across many node partitions.

4.2.3 Locking Semantics

When an range is inserted, all the current ranges in the current node partition are checked against available dimensions for overlapping ranges. If there is an overlap the decision is taken based on the locking semantics. Locking semantics based on both read and write locks are supported in the system. As in conventional locking systems, if two overlapping range are read ranges (read locks) then the requesting lock is granted. In granting read locks, if there is an exact read lock already stored in the node partition, the counter of the lock is increased or else new range is inserted as a separate range even if is an overlap with an existing lock. If the overlap is with a write region (write lock) the requesting lock is balanced until the existing lock is released. The order in which waited locks are granted is explain below. The counter of the write locks is always one meaning its an exclusive lock.

4.2.4 Fair Queueing vs FIFO

When a new lock request comes in, if there are no conflicting ranges existing, the lock is granted right away. Though this works faster, this could lead to starvation of range locks as there is always a high chance that an item lock gets ahead of a range lock which is already pending waiting for another item lock. Since chance of an conflict is high in range locks compared to item locks (since range locks cover a wider area), a workload with high number of range locks (i.e. range queries prompting for range locks) could have bad impact on the throughput because of starvation.

While the option is there to use a fair queueing in the locking if the current workload is more like a web workload with far less ranges requests compared to items requests, the systems also supports FIFO (First In First Out). In this scheme when a lock request comes in, it will be checked against both currently granted locks and against locks that currently being held before granting the permission. If the new request has a conflict with a lock from any of those sets, then the new lock is also pushed to the waiting queue and will be granted only after all conflicts are gone. When a lock is removed, it notifies all the locks that were waiting on it and the locks that have zero conflicts will go ahead and get the permission while other will wait on the remaining conflicts to be released. While FIFO has some overhead on extra waiting it will serve both items and range requests similarly and workload with considerable range requests will get benefited from this. All the experiments shown in the thesis are performed under the FIFO scheme.

4.2.5 Deadlock Detection

With locking in place deadlocks could be a common source for performance lagging. Thus early detection or prevention is a good feature to have. Modern DBMS [6] uses approaches like wait-for graphs to detect cycle of dependencies and release the transaction with a smaller working set (work done so far). The proposed system detect based on time-outs. Time-outs have some advantages over the detection mechanisms such as, in a complex workload with hundreds of concurrent users scanning through wait for graphs could be very expensive, if the contention is high it will eventually be timed-out even if there is no deadlock and passing one graph scan would not guarantee that the transaction will be a victim of another deadlock in the future and get terminated. Thus in the new system, transactions are rolled back if lock waiting time exceeds the given time out for the respective table.

4.2.6 Workload Profiling

In solely depending on time-outs, the crucial decision is to set the correct time-out values. A lower value would terminate most of the transactions lowering the final throughput and a transaction with high contention will be the usual victim. A higher value would make locks wait more time and it will eventually make CPU utilization low since the majority of the time is spent on waiting. It would again lower the final throughput.

To decide the correct time out value per table, workload is profiled against waiting times. Once the profiling is done, a given percentile will be selected as the decided time out. Profiling is done per node partition thus for each locking tree different node partitions could have different values based on the congestion. The final percentile to be picked from the profiled data set can be configured. Lower percentile will have higher throughput with low contention as most of the slow transaction are getting rolled back but have negative impact when the contention is high when higher wait times become normal. System offers to options to profile the workload once in the beginning or periodically profiles and decide new time out values.

Initial time-out value which is the same across all the tables can be defined based on the user requirements. In running TPCC our initial time out was set 2 seconds since most of the transactions should have a return time below 5 seconds. The initial value has a direct impact on the profiled values thus has to be carefully selected based on the needs. All the lock profiling, and timing outs is done by the memnode. The proxy is only responsible for traversing the hierarchy and sending the request to correct node partitions residing in memnodes.

4.3 Higher Level Architecture

Figure 4.4 shows the entire component architecture of the system. The index manager and the lock manager are the components described above. Transaction handler exposes the generic transactional key value based interface to the DBMS query processing units. Storage handler which is part of the query processing units bridges the conventional DBMS query processing unit with the new transactional key-value storage layer. The Transactional handler together with the index and the lock manager collectively are known as the proxy. Each three layers (i.e. Query processing, Proxy and memnodes) could technically reside in three different machines however there are certain optimizations being done with passing data back and forth if the proxy and the query processing units reside in the same machine. Detailed description for each component is given below.

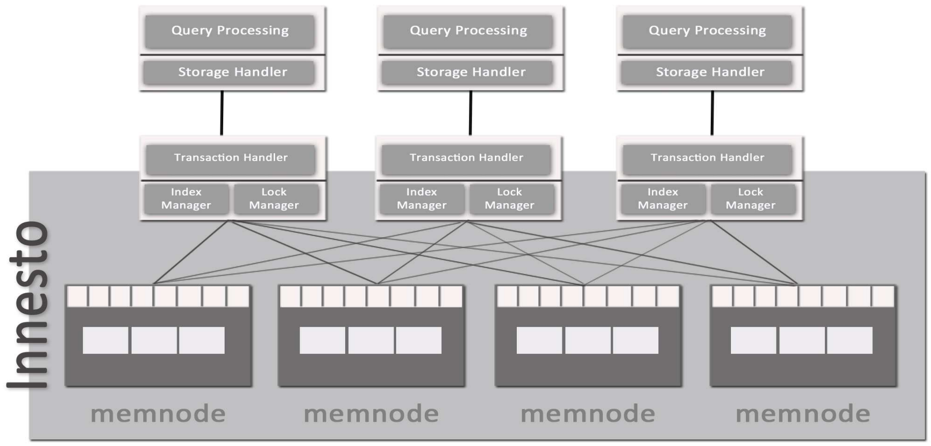

Figure 4.4: Full System Architecture

4.3.1 Index and Lock Manager, Memnodes

Core functionalities of the index and lock manager is given in the above sections. When it comes to the big picture, the index manager manager performs the CRUD (Create, Retrieve, Update, Delete)operations on the data stored in the memnodes. Transaction handler gives both the data and the key to be indexed when its an insertion or an update. If its a select or a delete transaction handler will only pass the index key to the index manager. The index manager has a key-value based interface and it is not aware about which index operations belongs to which transaction in the SQL level. Node partitions in the index have both keys and the respective data (both stored in the memnodes). Thus from the initial design, index manager acts both as the distributed index and the distributed storage.

The functionality of lock manager in the big picture is same as the index manager. The two main operations it supports are range insert and range deletion which correspond to the lock insert and lock release respectively. Lock insert could get returned with time out failure if the lock waiting exceeds the defined time out value. This will make the corresponding SQL level transaction rollback. As in the index manager, the lock manager too is unaware about SQL level transactions.

Memnodes are the only components which have a state and store data. It will save all the data in the memory thus avoid the overhead of disk dependency in serving proxy requests. Once the traversing is done in both the index and lock tree, the proxy will then connect with memnodes to communicate with the respective node partitions. And the communication is managed via light weight two phase commit. There are two types of memnodes in the system. All of the memnodes mentioned upto now are domain one memnodes which store node partitions from SPT. There are domain zero memnodes, which basically store meta data about tables (i.e, index and locking tree meta data for each table) and also have a distributed DHT to serve UNDO REDO logging for proxies (detailed description is given below on logging). In Figure 4.4 the small white boxes represent node partitions stored in memnodes.

4.3.2 Transaction Handler

The Transaction handler serves as the main interface to any outside query processing units. This can itself serve as a transactional key-value store with rich indexing and locking features. The main interface can be divided into three parts namely meta operations, data operations and transactional operations.

Meta operations

All the table creation and deletion comes under this group. Whenever a new table is created in query processing units, a respective table is created in Innesto. This meta data includes handler pointers to the index, to the locking tree and pointers to meta data such number of attributes, etc. All of these will be stored in the domain zero memnodes. Before any operation is performed on data the respective table-meta should be available in the system.

Data Operations

All the operations on the data come under this group. There are five different operations covering four CRUD sections. as given below.

- **Insert**: When a new record is inserted, the key to be indexed and the data portion will sent to the Transaction Handler from the QPU. When an insert is done, first the key is checked for duplicates and if found will be notified the QPU.

- **Lookup**: When there is a retrieval request and the exact key is available the request comes for a lookup with the key to be searched and the table giving the which index to be searched.

- **Query**: When a retrieval request is for a range of keys then the request comes as a range with min and max values to be included.

- **Update**: If there is a value to be updated then the Transaction Handler will receive the key and the value to be updated.

- **Delete**: If there is a value to be deleted then the Transaction Handler will receive the indexed key of the value to be removed from the index.

If the operation (insert, update or delete) is part of a transaction then the result will be buffered until the parent transaction is committed, or discarded if the transaction get rolled-back. Depending on the isolation levels assigned for the current transaction, a lock will be requested before each operation and will be released at the termination of the transaction. Currently only Serializable and Read Committed isolation levels are supported. In serializable both read and write items including ranges are locked until the end of the termination of a transaction. In read committed mode, write items are locked as usual till the end of the transaction but the read items do not acquire locks. Thus each time a read is done the same query might return different results within a transaction since a fresh read is done each time from the index.

If the lock is rejected due to time outs the entire operation is denied and the QPU will get the error message. When a data operation is failed, the all the related operations to the failed operation gets undone. However it is up to the QPU to decide whether to rollback the entire transaction or not. E.g. if a index insert is failed due to duplicate data, all the locks acquired for the insert will be undone and then report the failure to the QPU.

Transactional Operations

Apart from operations on the data, the system has to manage transactions as well in order to keep track of locks acquired and release them on time. There are three main operations coming under this group.

- **Transaction Start**: When a new transaction is starting QPU will notify the Transaction Handler, and it will start tracking the locks acquired and buffering required operations.
- **Transaction Commit**: When a transaction terminates successfully, it will first execute all the buffered operations and then release the acquired locks.
- **Transaction Rollback**: When a transaction terminated unsuccessfully, all the buffered operations are discarded and the acquired locks will be released.

Crash Consistency

If configured the proxy can run with a crash-protection mode. In that mode, all the operations executed under a given proxy will be pushed to a DHT as a UNDO log and when the proxy receives a commit request all the operations in that transaction will be pushed to a REDO log. UNDO logs contain only locks acquired during the transaction since index modification are only performed in the commit. The REDO log includes all the index modifications performed during the transaction and not yet executed on the actual index. Thus if a proxy gets crashed due to some reason, the proxy can look for the pending logs in the DHT (which is again an on-memory distributed storage) and act accordingly. The functionality of these logs is same as the conventional UNDO - REDO logs found in DBMS.

Apart from providing the protection for the proxy in an event of a crash, synchronous logging in the memnodes enables to recover data from a crash in the memnode as well. Though the memnode replication is provided in the original Sinfonia, Innesto currently does not provide replication and it is listed as the next step to be taken in the project.

User Management

Usually there are more than one proxy running in each machine to serve query processing units. There is one master proxy to which each new user connection from a QPU gets connected. The master proxy will then redirect the new connection to a different proxy based on round robin to balance the load. Each user connection from a QPU gets a new session in the proxy. This user model assumes the query

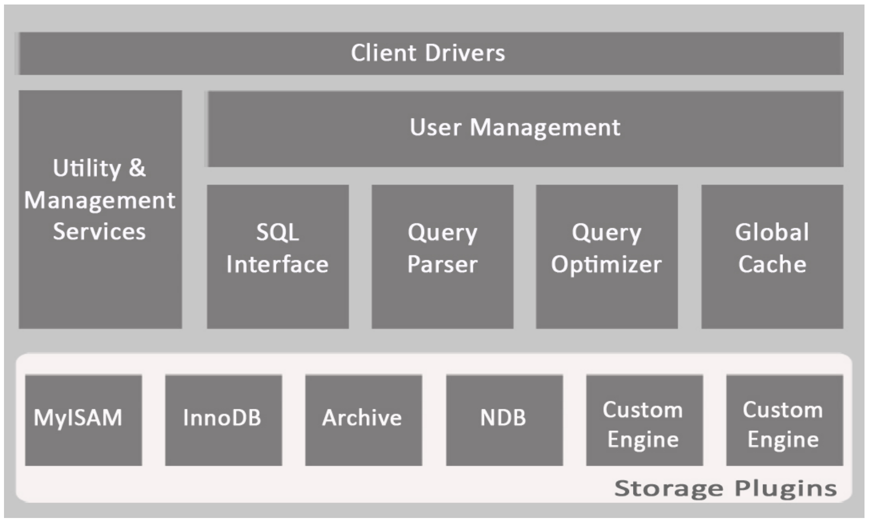

Figure 4.5: Current MySQL Architecture

processing units will work under thread-to-connection model so that proxy will get a new connection request each time a new user thread is spawned.

4.3.3 Query Processing unit

In the work presented in the paper, the implementation and the evaluation is done using MySQL DBMS engine [6]. The choice was made to use MySQL because of its wide spread usage across domains specially the usage as a hosted database service [1, 5]. Its storage plugin architecture made the integration easier and fail proof. MySQL storage plugin gives a well defined record based interface to the MySQL which matched perfectly with the Transaction Handler which also works on per record basis.

Figure 4.5 shows a further detailed architecture of MySQL. As its shown apart from the core functionalities mentioned in Section 2.1 , there are other components responsible for utility and management work such as logging, replication, recovery.

With multiple MySQL engines actively working at the same time, it is essential to balance the load among MySQL engines. Although the new system does not support any load balancing mechanism at the QPU level, any load balancing tool such as MySQL Proxy can be directly used to balance the load based on user congestion. This will become even more important when providing DBMS as a service.

4.3.4 Storage Handler

The Storage handler is responsible for mapping SQL based queries to respective key value based operations to be sent to the Transaction Handler. The plug-in architecture of MySQL made the storage handler more robust and less error prone as it already supports custom made storage handlers with a record driven interface. It sends out corresponding requests to the proxy based on the invoking method in MySQL. It is the storage handler that decides whether to do a lookup or range query and also decides on the key value range limits based on available data. The Storage handler as part of each data operation converts DBMS specific data types to a generic form.

To reduce the communication with the proxy, the storage handler caches all the returned results from the proxy and subsequent requests to the exact same data set will be served from the storage handler cache. Caching only happens in the isolation level ensures that the cached data will not be modified by others for the duration of the transaction. Cache items are modified accordingly based on operations performed on them enabling to return latest results on subsequent requests.

It is the storage handler than makes the rest of the system independent of the particular QPU being used. Since the storage handler is doing all the QPU specific conversion, technically multiple of QPU of different types could be running simultaneously. This is a crucial feature to meet the design requirement of general applicability. This also enables the system to leave the DBMS engines' complex query processing subsystems intact. Thus any MySQL user (or any DBMS that sits on top) can easily migrate to the new system with minimal hassle.

Chapter 5

Evaluation

5.1 Setup and Benchmark

All the experiments were performed on a cluster of commodity machines. Each has two quad-core Xeons (E5506) and 32 GB memory. Unless otherwise mentioned all the experiments we followed the following set-up. In each machine, we run the proxy, memnodes, MySQL server and the benchmark client. In each of those machines, there are 15 single threaded proxies and 5 single threaded memnodes. And on the same machine there are two MySQL servers serving user requests. Apart from one experiment, logging was disabled during the experiments and the data were only residing in the memory. Each MySQL sever is only communicating with the local proxy.

To emulate a practical workload on the system we used a benchmark client [9] which was based on the TPC-C [10] specification. The TPCC benchmark is a mixture of read-only and update / insert heavy transactions. The nature of the query set defined in TPCC covers a broad range of queries from complex range queries to simple lookup / update queries thus not only it covers majority of queries found in the real world and also its final out come gives a better picture about the underlying systems functionality. Since our main objective was to throttle the system as much as possible and to see how it scales none of the experiments were using wait times as specified in the TPCC specification. All the test numbers reported have the correct transaction mix and all the reported transactions are within correct latencies specified in the TPCC specification. All the transactions were executed in serializable isolation level except to stock-level transaction which ran in read-committed isolation level as permitted in the specification. Before each experiment, the system was loaded with data based on the TPCC spec. Whenever TPCC throughput is given its the number of New-Order Transactions per minute.

To compare the performance of iEngine against a standard DBMS we choose the InnoDB storage engine, the default engine in MySQL. All the experiments were using MySQL 5.5.8 source distribution and using InnoDb 1.1.4. InnoDB is the only storage engine shipping with MySQL that supports transactions and it is being used widely across many domains. It is also being used in couple of cloud hosted database offerings [1, 5]. InnoDb was disabled when MySQL uses iEngine as the storage engine. There were many configuration changes done to fine tune InnoDB to work in the current hardware among which buffer pool was configured to use 28G, log is flushed only once in a second using fdatasync, double-write was disabled and log file size was 1900M (For further configuration options see Appendix A).

The consistency checks given in the TPCC specification is used to verify the results of the TPCC outcome and of the functionality of iEngine. When iEngine is used as the storage engine, internal caches are disabled in MySQL since it could use stale data and has no information to verify it.

5.2 Strong Scaling

To evaluate how the system reacts to adding more query processing power for a fixed dataset (commonly known as *strong scaling*), we created a setup where three machines each running 8 single threaded memnodes are dedicated to store the data. We increase the number of machines from which we query through a benchmark client from one to three. Each client machine had 2 MySQL servers, and 24 client proxies. The total number of users were equally divided among each MySQL server. The data size was 90 WH stored across three machines.

Fig. 5.1 shows that cumulative throughput increases as we increase the number of machines querying the data stored in a fixed number of memnodes. At the same time the maximum number of concurrent users we can go before TPCC starts

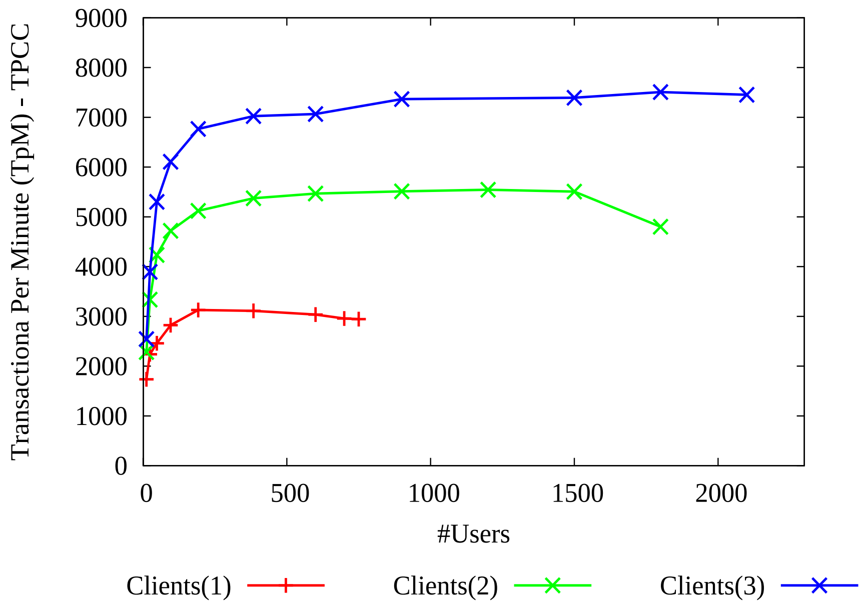

Figure 5.1: Strong Scaling in Query Processing

failing increases from 750 (querying from single machine) to 2100 (querying from three machines). This shows that we can add more query processing machines for a fixed size data set and increase the throughput. This is due to the fact that these client proxies and MySQL servers are entirely state-less enabling the user to add or remove them as required. This proves the scalability of the system on one dimension of adding more query processing units will gain more throughput.

5.3 Contention

We also evaluate how the system handles the high user contention compared with InnoDB. Usually the number of active users grows with the data size which is not harmful in the sense of contention. Contention happens when the user demand grows for a fixed or small set of data where everyone is interested in the same data portion similar to slashdot effect in web servers. This is quite common in the current web or for any other software appliance (e.g., in a retailer store this could be that every customer is interested in the special offer section only).

To emulate the above scenario, we increase the number of concurrent users using fixed number of machines for two fixed data sets (144 warehouses and 240 warehouses in two separate attempts). For both datasets we increase the number of users until TPCC starts failing. For the proposed system, it is a 6 machine setup where each machine has memnodes, proxies, MySQL servers and the benchmark client. For InnoDB, it is a single machine setup where both benchmark client and MySQL server runs on the same machine.

Figure 5.2 shows the impact of the increase in demand for a fixed size data set. For a smaller number of users InnoDB performs really well compared to the new system but as the demand increases InnoDB drops steeply and TPCC starts failing (at 850 users for 144 WH and at 1100 for 240 WH). Where as the proposed system can go upto 2700 users for 144 WH and 1950 for 240 WH. Figure 5.3 shows the number transactions executed per user. It shows how both systems degrade in that metric but demonstrates that iEngine persists further compared to InnoDB as the number of users increases.

We suspect that the steep drop down in InnoDB occurs due to the MVCC based concurrency model it follows where its susceptible to high contention. It outperforms conventional two phase locking in low user demands mainly due to its less overhead in MVCC however lags badly with high contention. This again reiterates that if we remove the bottleneck of a centralized setup in two phase locking, it can be used effectively and it handles contention quite well compared to MVCC.

5.4 Web Compatible Workloads

Though the TPCC workload emulates an online retailer, current web workloads are less stressful in terms of types of queries that they use. Thus different TPC benchmarks exist which more closely resemble web workloads with less cumbersome queries, the majority being simple lookups where as TPCC has considerable amount of range queries. In order to both resemble a simpler workload and to see how the systems behave in a write intensive workload we changed the benchmark client only to execute New-Order transaction which does not have any range queries. With this workload, 33% are inserts, 44% are lookups and 22% are updates thus it is a write intensive workload. Although for most of the systems, reads dominate the workload writes are equally important. In this workload, updates are

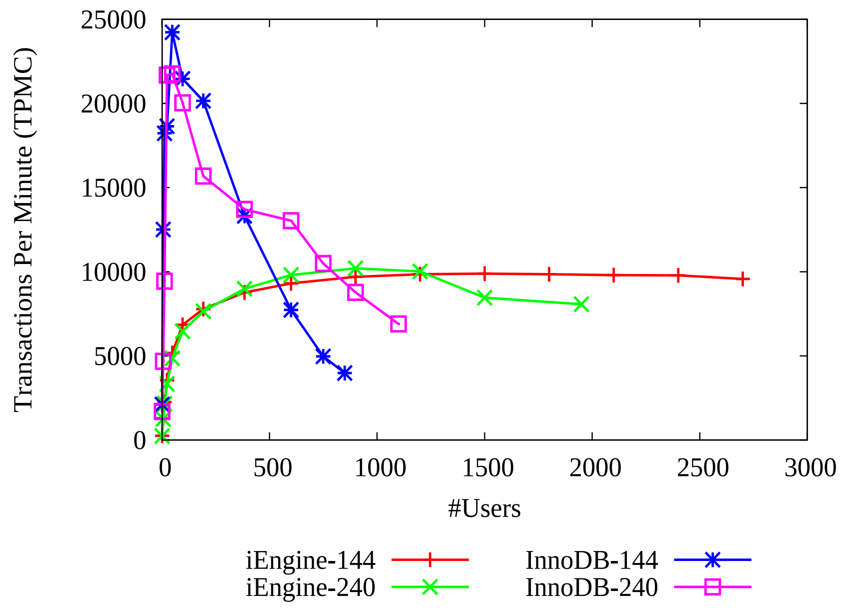

Figure 5.2: Scaling with Users

not changing any indexed keys only the value is getting changed.

The experimental setup is same as previously described where the new system is spread across six machines where as InnoDB is running in a single machine. We kept the data size (240 WH) and number of client machines fixed and increase the number of users to see how the system performs. The test was running under serializable isolation level.

As it is shown in Figure 5.4, following the same pattern InnoDB starts off well with smaller number of users and then falls steeply. On the other hand iEngine maintains the performance level and then gradually falls down as we increase the number of users. This pattern could be equally attributed to the concurrency control mechanism in InnoDB and by the BTree based indexing happening in InnoDB. This is an indication the SPT based indexing is a better alternative to conventional BTree based indexing as it can handle the contention better than InnoDB. This could also give an indication how the new system could out perform InnoDB in similar web workloads with a considerable portion of write requests.

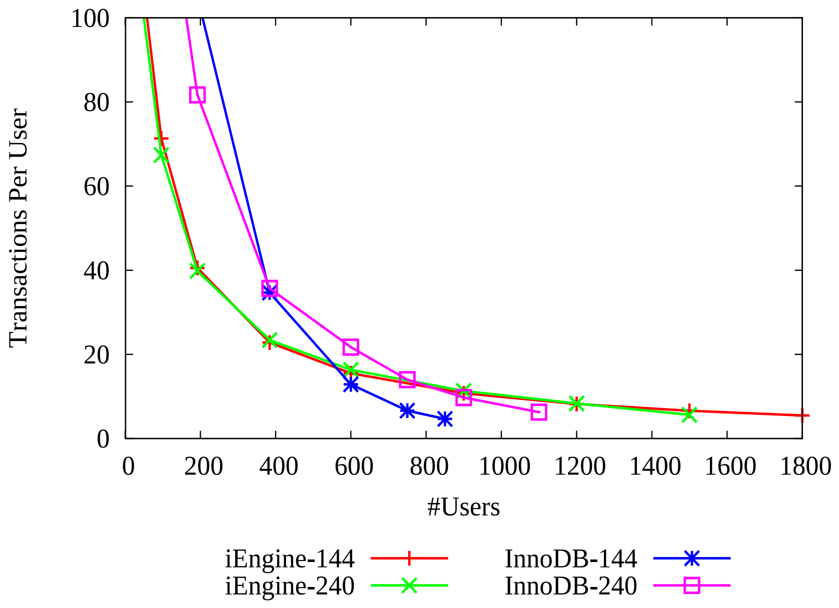

Figure 5.3: User Contention

5.5 Range Queries

With the current highly competitive environment most of the databases are used both in OLTP and OLAP type workloads to continuously improve the service to end users. Thus the ability of the database to serve both types of workloads effectively is crucial. Usually most OLAP type operations consisted of costly range queries and at the same time most of the distributed data storages suffer from lack of efficient support to serve them.

To evaluate how iEngine behaves in the presence of a light weight OLAP type workload, we use the Stock Level transaction in the TPCC specification. The Stock Level transaction consists of one OLAP type query and few lookups. As with the last two experiments, for fixed data size (240 WH) we increase the number of users to see how both InnoDB and iEngine scale in the presence of costly range queries. The experiment was performed under read committed isolation level as permitted by the TPCC specification. Thus no proxy level locking is involved in the new system. However each time a read is performed data is locked for the time of the operation in the memnodes.

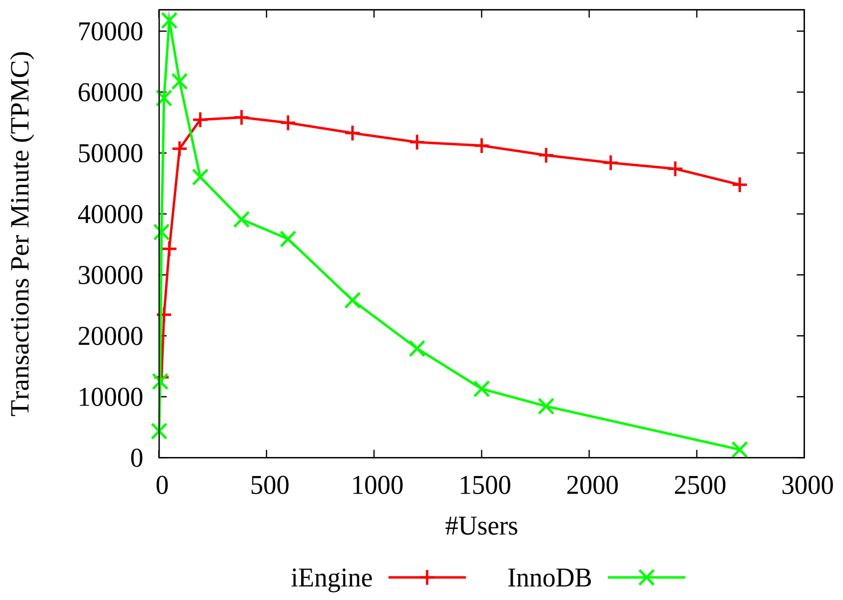

Figure 5.4: Web Compatible Workload

Figure 5.5 shows how differently each system responds to range queries in the presence of increasing demand. The iEngine out performs InnoDB in the number of Stock Level transactions it can execute per minute by order of magnitude. All the numbers reported on iEngine was under the permitted latency given in TPCC however as the number of users increased, percentage of transactions under correct latency falls from 100% to 0.6% in InnoDB. This could be due to the fact that InnoDB tries to keep a version of every read they do to provide the read isolation under MVCC.

5.6 Crash Consistency Cost

Although, for most of the businesses the entire data set can be fit into the memory, there can be a requirement for recoverability in the event of a crash, power failure, etc. The iEngine offers couple of options to persist the data as a precautionary action. It can be protected from proxy crashes (from UNDO and REDO logs) and from a memnode crash via synchronous logging. Crash consistency comes with a cost due its increased network communication and disk dependency. Here we

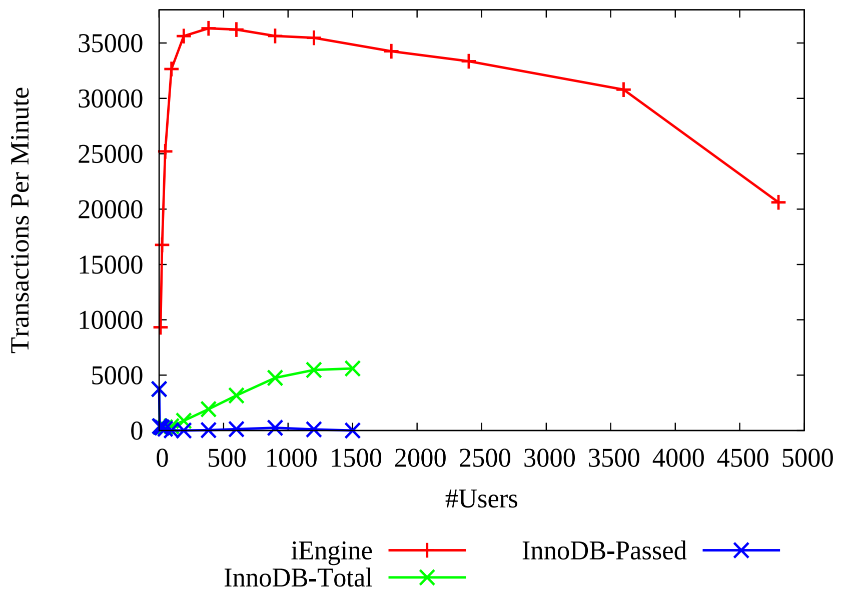

Figure 5.5: Range Queries

evaluate the cost by running a setup similar to previously shown experiment. Two dedicated machines run memnodes storing the data while four other machines do the querying. Experiment was conducted while the iEngine was configured to have both synchronous disk logging and transaction logging in the proxy is enabled. The initial data size was 60 WH.

As is shown in Figure 5.6, by running the application in the crash-consistent state it has performance reduction of 40% - 50% compared to running without the crash-consistent options enabled. The overhead also makes the system to start failing TPCC earlier than without logging enabled. While the current version only supports these two options, future versions will support more options to recover with less overhead See Section 7.2.

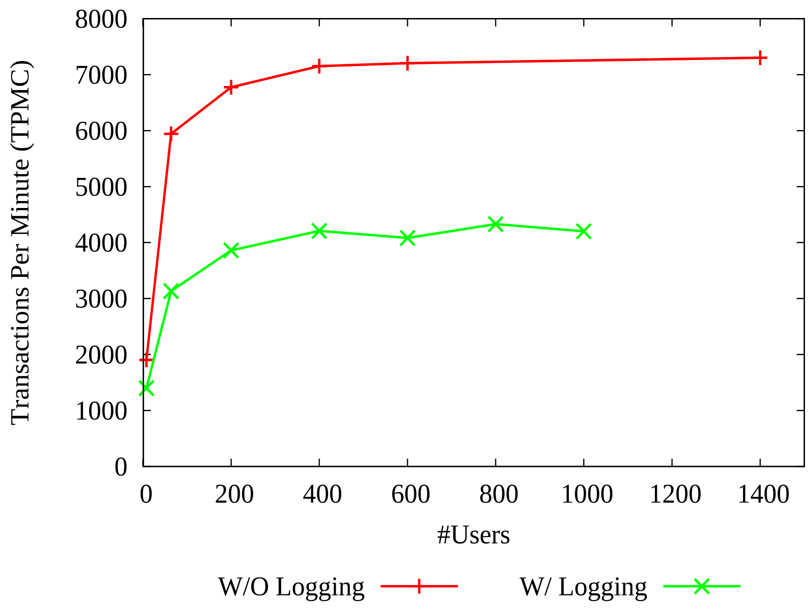

Figure 5.6: Cost of Crash Consistency

Chapter 6

Related Work

Scalable data storage has been one of the most explored directions in the recent past and there are lots of interesting new directions spawned as a result of that.

6.1 Data Partitioning

Partitioning has been the standard mechanism used in scaling out DBMS beyond one machine. Majority of current DBMS supports partitioning one way or another. With current complex workloads work has been shifted on new directions such as automatic partitioning based on workloads specially in case of providing DBMS as a service.

Schism [18] uses workload aware partitioning to minimize costly multi- partition distributed transactions. The current workload is periodically monitored to identify the set of tuples accessed together and uses graph partitioning to find balanced partitions reducing the number of cut edges (which represents a distributed transaction). Relational Cloud [19] uses the above partitioning scheme to decide on correct data placement across autonomous DBMS engines which are used as a back-end to a DB-as-a-service. The efficiency of the data placement thoroughly depends on how easy it would be to partition the workload reducing distributed transactions. The partitioning also gives the notion of a shared-nothing architecture giving each DBMS engine autonomy. H-Store [26] (Commercially known as VoltDB) is memory-resident database that partitions data across many single threaded engines. Though originally VoltDB could not dynamically scale out and scales in, [32] have shown that it can be done dynamically. However, VoltDB cannot be considered as a general purpose DB storage due its limitations in SQL semantics and the partition scheme has to be set upfront.

6.2 Key-Value Stores

With ever-increasing user demand and data size, industry response to even higher scalability requirements was data management systems with lesser expressibility and lesser consistency guarantees but with very high scalability and lower latency operations.

Data storage systems commonly known as NoSQL systems are designed in such a way to scale horizontally with no single-point bottleneck compared to conventional DBMS [3, 17, 22]. They usually use hash based data placement to load balance and spread the data across many nodes. While achieving high scalability they compromise higher consistency levels and rich query interface. There are different flavours in the query model such as column-oriented, document based, key-value, graph based, etc. Though some systems support a variant of indexing and range queries the majority do not have an efficient way to support such queries. At the same time isolation among operations is hard to achieve. While these are superior in certain domains they might not be good candidates for general use case and it has been shown lately that conventional DBMS can out perform these in certain workloads [20].

6.3 Scaling Storage

It did not take that long to realize that DB community moved faster than they should have to this new NoSQL and this sudden move has put tremendous pressure on the application development. Since most of the bulk carried out DBMS are not in the application layer which makes already complicated application logics even more complicated. Then the work started on coming to middle ground between two extremes of DBMS and NoSQL.

There have been efforts to re-architect the structure of conventional DBMS by de-coupling the storage layer from the query processing [14, 15, 21]. They have built database systems on top of a highly scalable storage system and have integrated the storage layer at the page-level or more granular data record level. Thus the storage can be scaled independently from the query processing components. However, in most of these systems a small group / partition is defined only in which strict consistency is guaranteed. Transactions reach beyond such a group have lesser consistency guarantees. The iEngine also has a similar architecture but it does not restrict full consistency to a defined set of boundaries. Strict consistency is achieved irrespective of the data being touched in a transaction.

6.4 Scalable Transactional Models

While the main focus is on re-modelling the DBMS architecture and new query models, there is an interesting body of work being done on finding ways to implement transactions in a DBMS in a more scalable manner.

DORA [33] explores how to implement threads-data model rather than thread-transaction model to reduce the contention. Dueteronomy [29, 31] talks about an architecture similar to the iEngine where there is a clear separation in the data handling and transaction handling components in the DBMS. Consistency Rationing [27] explores how we can adaptively change the required consistency level as and when it matters to get better performance and cost effectiveness in the cloud. There is another notable work being done on new concurrency control mechanism to scale in the presence of high contention which is based on transactional dependency [28].

6.5 Distributed Indexing

There are similar work being carried out on scalable indexing based on distributed data structures. Most closest work is done on Distributed Btree [13] which suffers in the presence of insert heavy workloads. Minuet [35] is the improved version which supports OLAP and OLTP based queries but only supports simple transactions. Both versions run on top of Sinfonia [12]. However they still have an abstract data model compared to iEngine and multi-dimension feature in SPT helps to have more finer level locks. Another work is proposed to use BATON [25] as an overlay over a cloud storage for efficient range queries [39]. However they have a scalability bottleneck since they serialize all transactions in a single node and provide optimistic concurrency control which suffers from high contention as shown in the evaluation.

6.6 Distributed Locking

Specialized locking systems, such as Chubby [16] and ZooKeeper [24], provide scalable locking along with strong consistency guarantees. Due to the way locking is implemented in these systems, efficiently implementing complex multi-attribute range locking (which is a common requirement of an RDBMS) is fundamentally not feasible. Google's Percolator [34] also implements a scalable locking mechanism on top of BigTable, but only works on snapshot isolation and can not support serialization isolation, as required by strict consistency. Thus these specialized systems lack the generality and expressibility needed by a RDBMS. The distributed range locking offered in iEngine is very effective in a distributed setup and unique as well.

6.7 Commercial Offerings

To keep up with the demand there are quite a few commercial services that offer DBMS as a service in the cloud [1, 2, 5]. Main issues with these services are they are either not capable of scaling beyond one machine or not providing full consistency in the presence of distributed transactions. There are dozens more DB products that offer products that can be deployed in-house in a private cloud or can directly obtain the functionality from a hosted service in a public cloud [4, 7, 11]. The majority of these systems use MVCC as a concurrency control mechanism and suffer a lot in a high contention scenario. While the conventional two phase locking is a bottleneck in scaling, with a novel distributed approach, iEngine has shown it can withstand 3X-4X more user contention than current systems.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, a new data storage architecture is presented which is based on a Scalable, Distributed Spatial Partitioning Tree. Our experimental results show that such an architecture should be a practical alternative to the current storage systems. Locking and indexing are two core components in the storage that poses a huge bottleneck in scaling a DBMS. Distributed SPT provides a practical alternative to those components while providing all the features provided in conventional systems.

Most of the scalable systems compromise the generality of data storage by designing to suit certain domains. The iEngine architecture presented in the thesis can be used in any domain that was using SQL for its data management requirements. In fact the new architecture could actually support different DBMS front ends at the same time. This makes the migration to the new system quite smooth and hassle free.

In modern scalable systems, they scale only in the presence of simple web type workloads consisting simple item lookups and inserts. In the results shown in the evaluation, it shows that iEngine can scale even in the presence of the simple web type workloads as well as with complex range operations. This is becoming a crucial aspect of data storage systems since more and more users are now using both OLTP and light weight OLAP in their databases.

Partitioning has been the de-facto mechanism to scale out beyond one machine. However with distributed spatial partitioning approach as the nodes get congested it will automatically distribute data across available nodes thus no manual partitioning is required. This provides a new dimension as with complex workload patterns partitioning is not trivial.

With novel logical locking based on SPT, the system can scale beyond a single machine and still provides strict isolation and consistency. This is a crucial feature to have since its now been understood that though lesser consistency models give higher scalability, it makes the application more complex and error-prone.

7.2 Future Work

Live Node Migration

Use of SPT makes live node migration easier with migrating live node partitions residing in memnode to another. This will provide the much needed scale out and scale in features which is a crucial feature to have in a cloud style deployments.

Dependency Aware Concurrency Model

The current optimistic concurrency model assumes no conflicts on touched data regions. The assumption becomes less and less practical with high contention systems. In this new model (where some early work has been already done) the assumption is made on the completion of a transaction which is going to be more practical in a high contention systems. Adopting such a model would further increase the performance.

Multi Tenancy

In providing DBMS-as-a-service, multi tenancy is a must, iEngine does provide multi tenancy in terms of giving a different indexing and locking each different user, thus making sure that a given users only sees the data belongs to that user. However providing more privacy measures would make the system more practical for multi tenancy.

Data-as-a-Service

Providing Data-as-a-service is gaining popularity allowing people to use real world data to carry out numerous activities such as market research. With iEngine using as a DBMS-as-a-service, it gives the luxury of having real world data from different clients. Given the permission is granted by each client we could share those real world data with the 3rd party clients with restricted access.

Data Replication

Current iEngine does not provide any type of replication. However the architecture of iEngine provides the ability of implement replication in two levels. At the proxy level we could replicate the proxy to provide fall back option in the event of a crash of the proxy. Node replication in the memnode could provide both protection from a crash and more performance in read operations.

Bibliography

[1] Amazon rds. http://aws.amazon.com/rds. → pages 1, 25, 28, 39

[2] Microsoft azure. https://www.windowsazure.com/en-us/home/features/data-management/. → pages 1, 39

[3] Apache cassandra. http://cassandra.apache.org. → pages 37, 51

[4] Clustrix. http://www.clustrix.com. → pages 39

[5] Google cloud sql. https://developers.google.com/cloud-sql/. → pages 1, 25, 28, 39

[6] Mysql. http://www.mysql.com/. → pages 19, 25

[7] Nuodb. http://www.nuodb.com/. → pages 39

[8] Three dimensional spt. http://en.wikipedia.org/wiki/Octree. → pages 9

[9] Transaction processing performance council. https://code.launchpad.net/ percona-dev/perconatools/tpcc-mysql. → pages 27

[10] Transaction processing performance council. http://tpc.org/tpcc/default.asp. → pages 27

[11] Xeround. http://www.xeround.com. → pages 39

[12] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174. ACM, 2007. → pages 3, 8, 38, 48

[13] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1:598–609, Aug. 2008. ISSN 2150-8097. doi:http://dx.doi.org/10.1145/1453856.1453922. URL http://dx.doi.org/10.1145/1453856.1453922. → pages 15, 38

[14] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research (CIDR)*, 2011. → pages 2, 37

[15] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 251–264, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi:http://doi.acm.org/10.1145/1376616.1376645. URL http://doi.acm.org/10.1145/1376616.1376645. → pages 37

[16] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX, 2006. → pages 39

[17] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2): 4, 2008. → pages 2, 37, 51

[18] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010. ISSN 2150-8097. → pages 36

[19] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: a database service for the cloud. In *In Proceedings of 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011. → pages 1, 36

[20] C. Curino, E. Jones, Y. Zhang, and S. Madden. can the elephant handle nosql onslaught. *Proceedings of the VLDB Endowment*, 2012. → pages 37

[21] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic transactional data store in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, page 7. USENIX Association, 2009. → pages 37

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon’s highly available key-value store. In *Proc. of 21st ACM SIGOPS symposium on Operating systems principles*, SOSP ’07, pages 205–220, New York, USA, 2007. ISBN 978-1-59593-591-5. doi:http://doi.acm.org/10.1145/1294261.1294281. URL http://doi.acm.org/10.1145/1294261.1294281. → pages 2, 37

[23] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM, 2008. → pages 3, 4, 7

[24] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proc. of the 2010 USENIX annual technical conference*, pages 11–11. USENIX. → pages 39

[25] H. Jagadish, B. Ooi, and Q. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st international conference on Very large data bases*, pages 661–672. VLDB Endowment, 2005. → pages 38

[26] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1:1496–1499, August 2008. ISSN 2150-8097. doi:http://dx.doi.org/10.1145/1454159.1454211. URL http://dx.doi.org/10.1145/1454159.1454211. → pages 36

[27] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009. → pages 38

[28] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2095686.2095689. → pages 38

[29] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction Support for Cloud Data. In *Proc. CIDR*, 2011. → pages 38

[30] L. Lin, V. Lychagina, and M. Wong. Tenzing a sql implementation on the mapreduce framework. *Proceedings of the VLDB Endowment*, 4(12): 1318–1327, 2011. → pages 5

[31] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling. Unbundling transaction services in the cloud. *Arxiv preprint arXiv:0909.1768*, 2009. → pages 38

[32] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson. Elastic scale-out for partition-based database systems. In *Proc. International Conference on Data Engineering Workshops, Workshop on Self-Managing Database Systems (SMDB'12)*, 2012. → pages 1, 37

[33] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment*, 3(1-2): 928–939, 2010. → pages 38

[34] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. 9th Usenix Symp. Operating Systems Design and Implementation*, pages 251–265. → pages 39

[35] B. Sowell, W. Golab, and M. A. Shah. Minuet: a scalable distributed multiversion b-tree. *Proc. VLDB Endow.*, 5(9):884–895, May 2012. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2311906.2311915. → pages 38

[36] M. Tayarani Najaran and C. Krasic. SinfoniaEx : Fault-Tolerant Distributed Transactional Memory. Technical report, University of British Columbia, Department of Computer Science, 03 2011. → pages 48

[37] M. Tayarani Najaran, C. Krasic, and N. C. Hutchinson. Sinextree : Scalable multi-attribute queries through distributed spatial partitioning. Technical report, University of British Columbia, Department of Computer Science, 07 2011. → pages 48

[38] M. B. Uddin, B. He, and R. Sion. Cloud performance benchmark series, amazon relational database service (rds) tpc-c benchmark. Technical report, Stony Brook Network Security and Applied Cryptography Lab. → pages 1

[39] H. Vo, C. Chen, and B. Ooi. Towards elastic transactional cloud storage with range query support. *Proceedings of the VLDB Endowment*, 3(1-2): 506–514, 2010. → pages 38

Appendix A

InnoDb Configuration

This is the configuration used with InnoDB while performing tests given in the evaluation.

```
——transaction–isolation=serializable
——max_connections=2000
——max_prepared_stmt_count=232784
——back_log=2000
——thread_cache=100
——innodb_buffer_pool_size=28G
——innodb_io_capacity=4000
——innodb_read_io_threads=10
——innodb_write_io_threads=10
——innodb_buffer_pool_instances=8
——innodb_concurrency_tickets=2000
——innodb_additional_mem_pool_size=160M
——innodb_log_file_size=1900M
——innodb_log_buffer_size=8M
——query_cache_size=200M
——thread_concurrency=16
——table_cache=10000
——innodb_purge_threads=1
——innodb_log_files_in_group=2
——innodb_file_per_table=1
——innodb_max_dirty_pages_pct=90
——innodb_flush_log_at_trx_commit=2
——skip–name–resolve
——skip–innodb–doublewrite
——innodb_lock_wait_timeout=2
——query_cache_limit=6M
```

Appendix B

SinfoniaEx

Before moving to the Innesto based system which is inspired by Sinfonia [12], an earlier system was using something called SinExTree [37] which is again a Spatial Partition Tree which is running on top of SinfoniaEx [36] an open source version of Sinfonia. Sinfonia from the design has an optimistic concurrency model which forces mini-transaction to rollback and retry whenever there is a conflict. In the original design, memnodes do not have any knowledge on the data they store and their main function was to expose memory range to be used by proxies transactionally.

When the SinExTree was used, the architecture had three separate components, namely; Index manager, Locking manager and DHT based distributed storage. Index only had the key and the DHT key as the respective data using which we can retrieve the data from the DHT. For each operation the index is scanned first and using the result set each data item is fetched from the DHT incurring lots of overhead.

With SinfoniaEx, the minitransaction was following its original design of compare items thus data was also cached in the proxy. However as with the original design these minitransactions were rolling back as soon as there is a conflict. With locking based on SinExTree, whenever there is a lock conflict the lock request was returned with an error. Since there was no queueing mechanism in the older version of memnodes, retrying lock requests never guarantees that the lock will be granted. Thus this leads to a huge number of rollbacks in the MySQL level trans-

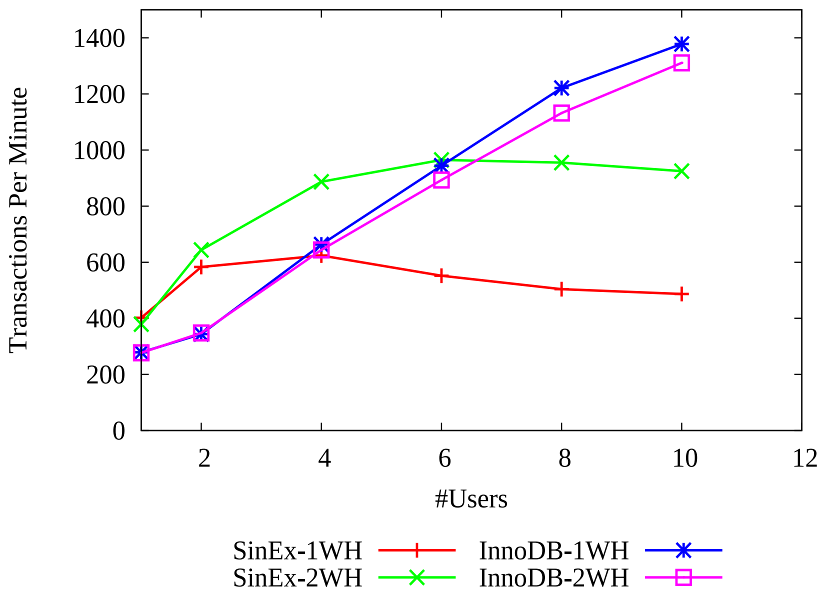

Figure B.1: SinExTree Thrashing with High Contention

action as we increase the number of users. Thus the system was thrashing with a higher number of users. As shown in Figure B.1 while InnoDB continues to grow with number of users SinExTree starts failing after 10 users since some transaction keeps on getting rolled back.

However the SinExTree was performing well if it is used only for indexing. It was not sufficient enough to keep the system from thrashing. Figure B.2 shows how SinExTree was scaling across machines for the TPCC workload as an indexing sub system.

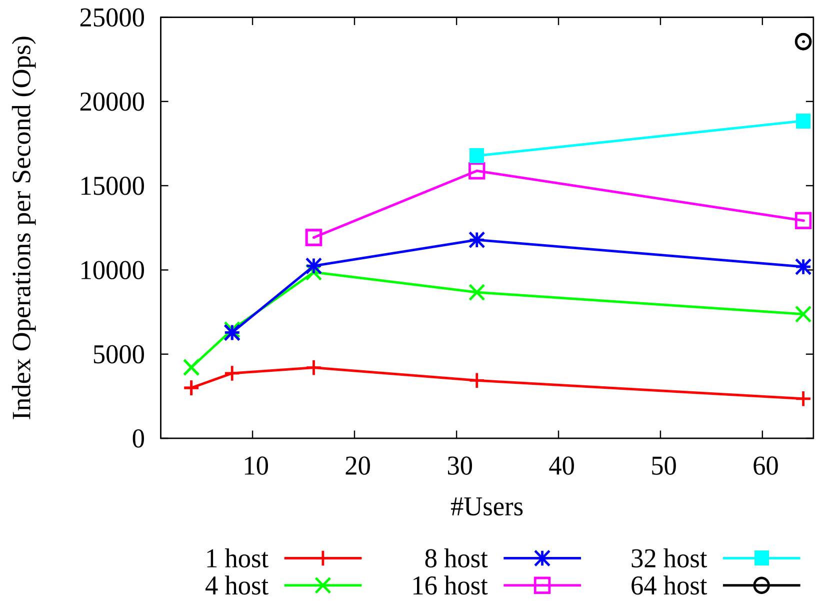

Figure B.2: SinExTree as an Index

Appendix C

MySQL over Cassandra

When the project was initially started the implementation choice was Cassandra [3]. Because of its scalability it was good candidate to start with. Cassandra has the column family model inspired by BigTable [17]. As with most other key-value based systems, Cassandra was using an eventual consistency model to scale out and to have availability over consistency in the event of a node partition. However it has the option for a fully consistent model as well based on quorum voting.

Locking

With zero isolation support from Cassandra, providing locking or any sort of isolation is a huge challenge. In the case of a single node MySQL setup, a data-store like Berkeley DB could provide logical locking and it works. The issue becomes non-trivial as soon as the locking goes distributed across machines.

Because isolation based on locking seems to be hard, lock-less isolation was the only option so Multi Version Concurrency Control was implemented on top of Cassandra. Each update or write triggers a new version. For each row there are different columns corresponds to a different version. When a table is first accessed, a transaction knows the highest version number and subsequent reads will only return columns with less than or equal to above version number. While this does not provide serializability, it provides snapshot isolation.

Indexing

Cassandra does provide the hash based indexing out of the box. While the default data placement strategy provided good load / congestion balancing, it does not provide features like range queries at all. However with byteordered partitioning data placement will be done in order thus range queries is not impossible yet very inefficient compared to the conventional tree-based range query approach.

With both lack of suitable setup for isolation and for the rich query model, Cassandra was not a good choice to move ahead with the integration of MySQL.