On the Design and Scalability of Distributed Shared-Data Databases

Group No 15

Amit Thakral (A20343683)

Kedar Kaushikkar (A20355218)

Pooja Mankani (A20354444)

Contents

Abstract

Partitioning data across many database instances has restrictions, like they are quite inflexible during large scale deployments. This paper has introduced an alternative architecture design for distributed relational database to overcome partitioned databases limitations. The architecture includes: decoupling query processing and transaction management from data storage tasks, sharing data across query processing nodes. However, this new architecture has a drawback of synchronization overhead during sharing of data across multiple nodes. So the paper has described ways to access data efficiently, concurrency control and the data buffering.

Introduction and Necessity of shared data Databases

This paper has introduced an architecture for distributed transaction processing, keeping in mind about operational flexibility. As the name suggests operational flexibility helps application to be elastic, give ability to grow or shrink the system as per demand, the ability to write and execute any kind of query, deployment flexibility and so on.

The architecture is built to keep RDBMS strengths which are ACID transactions, SQL (Structured Query Language). The two fundamental principles of the architecture are:

- The data is shared across all of the database instances. The benefit here is that the database instances can access the whole data and can execute any query over the database.
- Decouple the query processing and transaction management tasks from data storage. When compared to traditional RDBMS, these two are tightly coupled. However, here this separation facilitates elasticity and deployment flexibility.

The focus of this paper is on OLTP (Online Transaction Processing) workloads also. In all, the three contributions of the paper are as follows:

- Architecture design in distributed relational databases that provides fault-tolerance, scalability without any assumptions on workload.
- Techniques for efficient and consistent data access and this is shown using implementation of the shared-data architecture, a database system called Tell
- Comparing the new design and evaluate it against VoltDB, MySQL Cluster and FoundationDB.

Applying both of the fundamental principles used in the results in a two-tier architecture in which database instances operate on top of a shared store.

Shared Data Architecture

Design principles

- Shared Data:
    - The data accessible by every database instance and modified and the one that has no exclusive data ownership.
    - Advantages of shared data:
        - No complex DB Cluster
        - No partitioning reflected with the application logic hence simplified DB.

- Disadvantages of Shared Data
  - Updates in the Data need to synchronize.

- Decoupling of Query Processing and Storage:
  - The shared-data architecture is decomposed into two logically independent layers, transactional query processing and data storage.
  - This section summarizes about the fundamental difference between communication pattern between Shared data databases and Partitioned Databases.
  - In Partitioned Databases, data partitioning is not as performance critical as the data location does not determine where the queries need to be executed.
  - Instead the processing layer accesses records independent of storage Nodes they are located.
  - Shared Data architecture can be broken down into two independent layers
    - Transactional Query Processing
    - Data Storage: This is autonomous i.e. it is implemented as a self-contained system managing data distribution.
    - The Storage System acts as a distributed record manager that consists of multiple Storage Nodes.
  - Hence, replication and data re-distribution tasks are executed in the background without the processing layer being involved

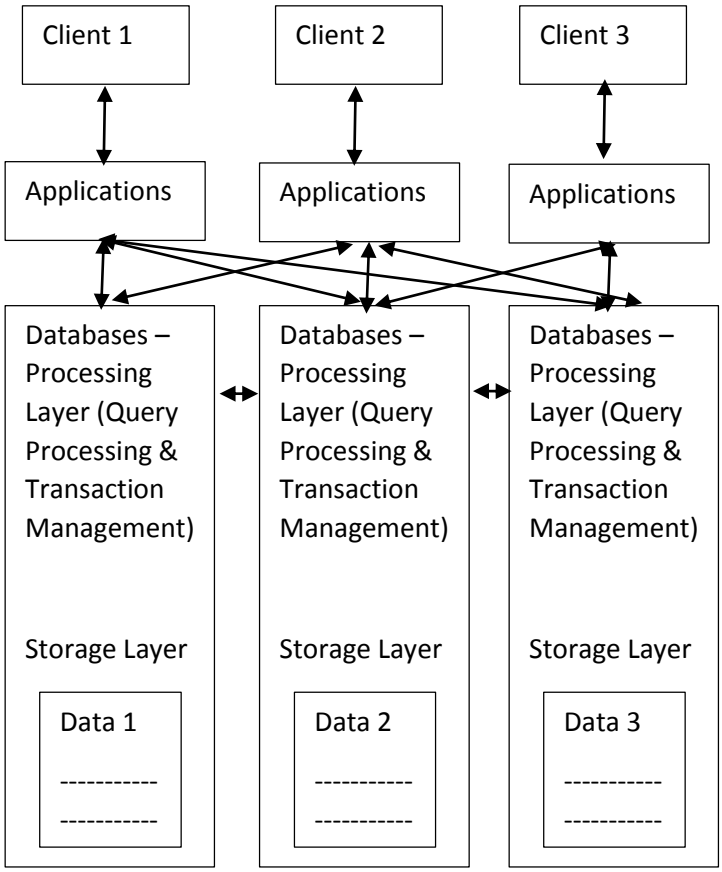

Fig 1. Partitioned Database

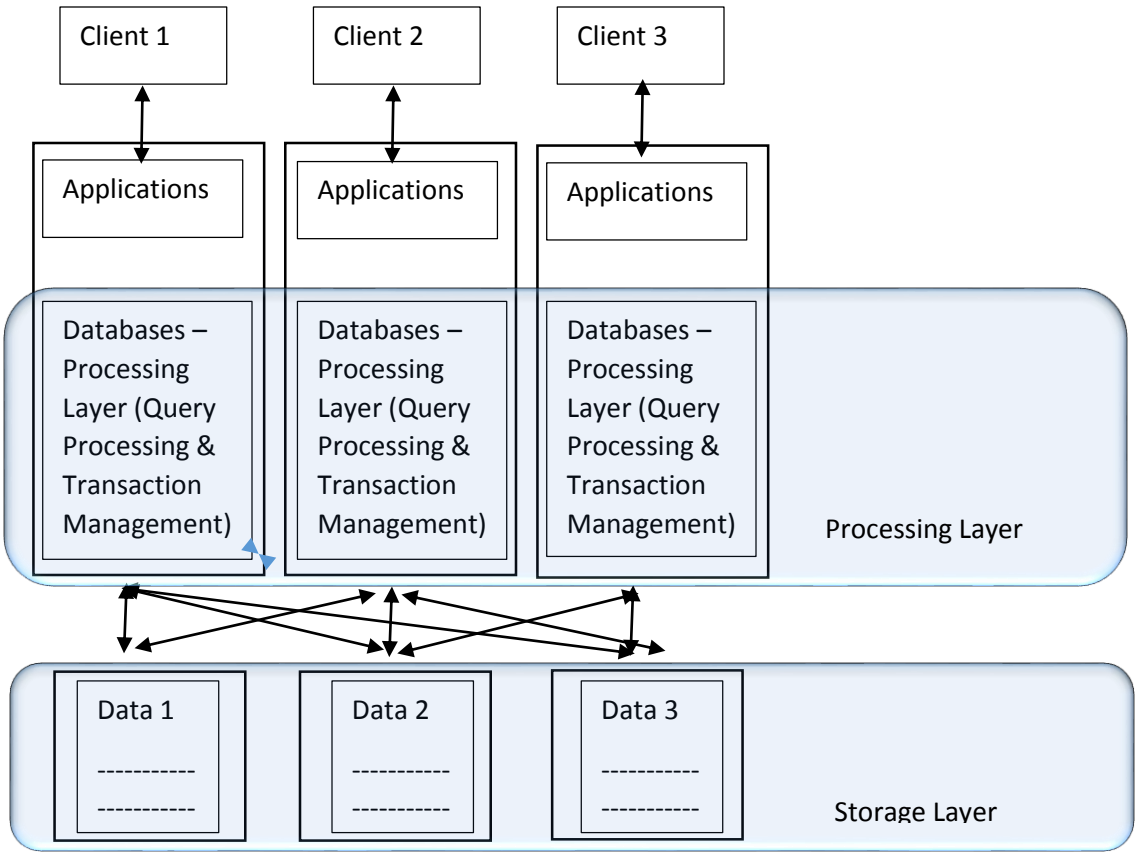

Fig 2. Shared-data Database

- In Memory Storage
  - The advantages of In-Memory Storage are:
    - Provides Low Latency
    - Avoids complex buffering mechanisms
  - However one of the concerns are that the Dynamic Random Access Memory being volatile can cause data loss in case of failures which must be prevented.
- ACID Transactions
  - A set of properties that guarantee that database transactions are processed reliably.
  - The transactions ensure isolated execution of concurrent operations and maintain data integrity.
  - Data is less prone to getting corrupted and ACID Transactions can be performed without limitation on the whole data.
  - This part of transaction management is a part of processing layer and does not make any assumption on data access and storage methods.
- Complex Queries
  - The complex queries do not cause any overhead or problems to the system scalability. The Data in shared-data architecture is logically separated from query processing.

- Data is shipped to the query which is done by processing nodes that retrieve the records required to execute the query.

Technical Challenges

- Data Access:
  - Data in shared environment is mostly located in remote location and has to be accessed over the network. Hence the in order to constantly keep updating the processing nodes with modified data , caching techniques are used but even caching is possible to limited extent only.
  - Hence for consistency, most requests take up data from the latest record version remotely from the storage layer.
  - One major challenge in this context is the correct scale or level of detail present in set of data storage.
  - The topic talks about storing data at a granularity of record as this provides a good trade-off between number of network requests and amount of traffic generated.
- Second major challenge is the shared data relating to data access paths and indexing. Compared with respect to data indexes are also shared across multiple processing nodes and hence can be concurrently modified from distributed locations. Shared data architecture requires distributed indexes that are highly scalable (distributed B+ Tree)
- Concurrency Control:
  - Concurrency control is required across all the Processing Nodes as any one of them can update the records in shared data architecture. Hence Concurrency control is required across all the Processing Nodes.
  - Assuming network latency is more than the update time, the data update latencies may due to more write conflicts resulting in higher abort rates.
  - Hence any mechanism for concurrency control that minimizes the overhead of distribution can result for highest scalability.

Limitations

- The efficient performance of this type of architecture can be presented on LAN because
  - Allows Low latency Communication
  - This is critical performance factor because as shared data provided limited buffering , it might result to heavy data access over the network
  - WAN is unsuitable for this type of architecture as the communication cost is more than LAN
  - Network saturation can occur due to heavy load or large records because of constant exchange of data between Processing and storage layer.

Tell as a shared architecture

Major Components of Tell

- Processing Node (PN): This is the central component that processes queries and transactions. These interact with the commit manager that takes care of the global query processing.
- Management Node: This helps in failure recovery and continuously monitors the system.
- Storage Node: This helps in storing the data. It manages the data records, indexes and undo log for recovery. In order to have transaction processing, get and put operations on single records are supported. This is a major component since Tell completely depends on the storage manager to process queries.

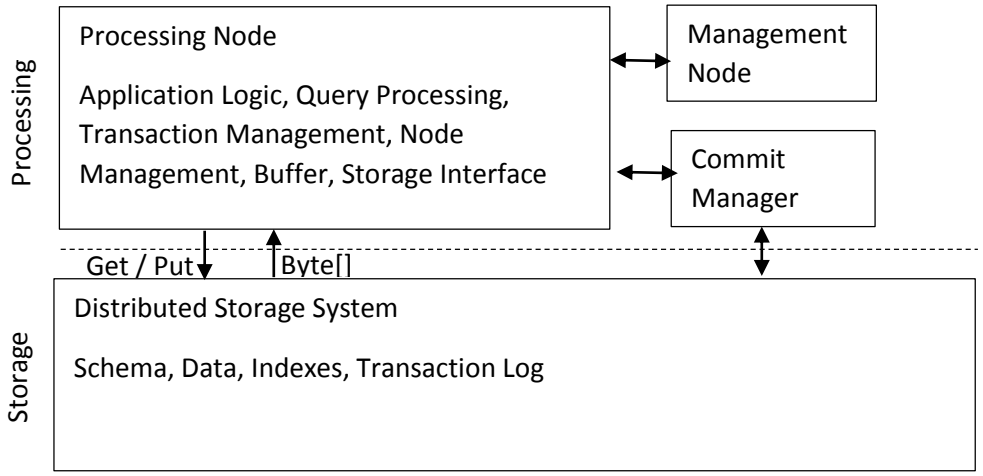

Fig 3. Components in Tell architecture

Concurrency control

- Concurrency control allows the transactions to run in parallel on multiple processing nodes (PNs) without hampering data integrity. Snapshot isolation is a protocol used in Tell to that guarantees ACID properties and always allows transactions to make progress.

Distributed Snapshot Isolation

- This is a multiversion concurrency control protocol that is implemented in multiple databases. Multiple version of every data item are stored in the database. A new version is created every time an update is made to the item. As a transaction starts all the versions of the data item is retrieved as a list. Only those version are retrieved that were written by finished transactions. This helps to maintain the consistency of the system. The updated are buffered and applied to the shared system during commit.
- A transaction T1 is considered to be successful if none of the items that T1 is accessing is changed externally by another transaction (Tn) since T1 has started.
- In case two transactions (T1 and T2) run in parallel two scenarios can occur:
    - T2 writes the changed item to the shared store before T1 reads the same item. In this case T1 will experience a conflict since the item is updated to the newer version.

  - T1 reads the item before T2 changes write the same item. In this case, T1 must be able to detect the new version once T2 has committed the changes and before T1 needs to write the changes.
- Thus for these cases, Tell executes a Load Link / Store conditional (LL / SC) operation in the storage layer. This is a pair of instructions that reads a value and the same can be updated only if it has not been changed in the meantime.
- This operations is more powerful that compare-and-swap since this solves the ABA problem. ABA problem basically occurs when one thread reads a data item twice and finds out that nothing is changed, however in between the two reads the item was accessed by another thread, updated and updated back to the original value. These updates in between can be easily detected by LL/SC operations. That is a write on a modified item fails even when the value matches the original value.
- In case all updates can be applied as required, we can conclude that there are no conflicts in the system. In case any conflicts occur transaction T1 will abort.
- SI avoids many update, insert and delete anomalies in the database. However, write anomalies make prevent it from making SI serializable. For making the snapshot isolation work as required, each node interacts with commit manager.

Commit Manager

- It manages the global snapshot information and enables transactions to retrieve three elements: Transaction ID, Snapshot descriptor and lowest active version number.
- Transaction ID uniquely identify a transaction. It is required before executing an operation. It also allows to define the version number for updated data items. It basically creates new versions for updated data items with the TID as the version number. Since TIDs are incremented version number are increased accordingly.
- Snapshot descriptor: It is a data structure that helps to specify which version of the data item needs to be accessed. It consists of a base version and a set of newly committed TIDs that are greater than b which signifies that these need to be committed. However, once committed the b would be b+1 and the process would continue.
- Lowest active version number (lav): It is the highest version number globally visible to all the transactions. All the version number smaller than the lav are candidates for garbage collection.

Transaction life cycle

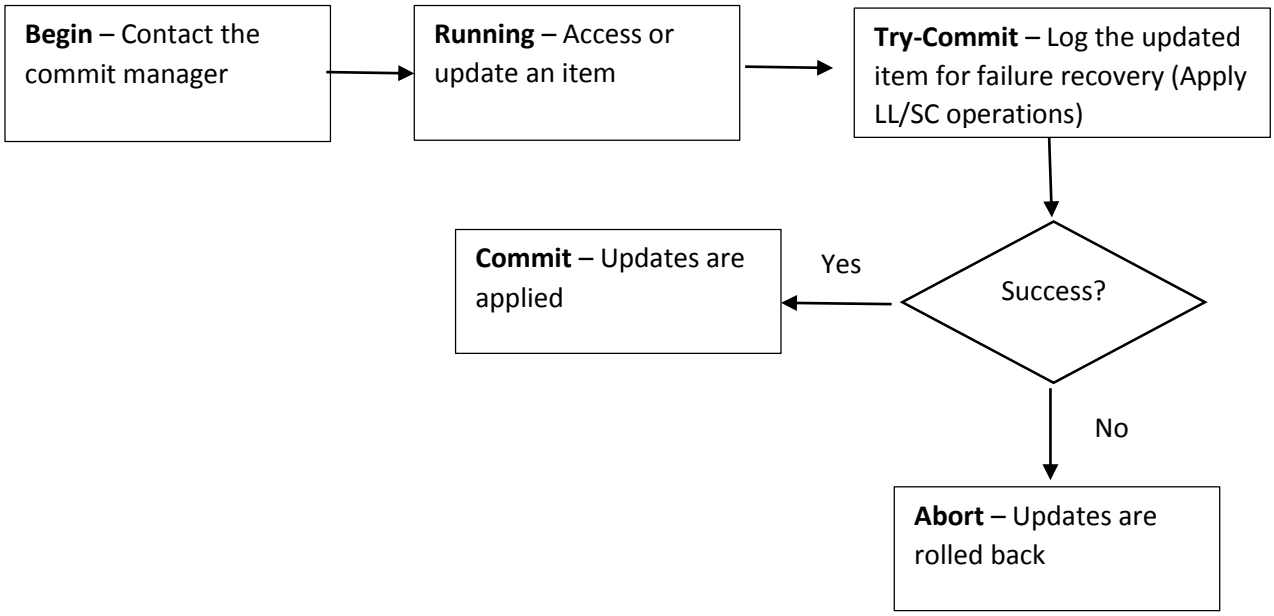

Fig 4. Transaction life cycle

Recovery and fail over

- Failure of a processing node: PN works on a crash stop model. As soon as a failure is detected the all the active transactions are aborted. A recovery process is initiated to roll back the active transactions of the failed node. The transaction log is used in this case. It is an ordered map of log entries located in the system. When the recovery process is started the active transactions are detected. Highest TID is retrieved from the commit manager and iterated backwards till the lowest active version number is retrieved. The lowest active version number acts as a checkpoint.
- Failure of a storage node: Storage node failure must be handled transparently. Since PN use SN for data access; failure of storage node must not lead to data loss. There should be a minimized downtime in case a storage node fails. Availability of the data is guaranteed by replicating the data. Data is always replicated before it is been accessed. In case the storage node fails the data is retrieved from the replicas. Management node in the storage system is used to detect failures.
- Failure of a commit manager: In case a single commit manager is available failure of the same will have wide system impact. PNs are blocked until the commit manager is available. Once all active transactions are committed, a new commit manager can be started. To state can be recovered from the last used TID. In order to prevent single point of failure, multiple commit managers are present in the cluster. Their states are synchronized. They write their state to the storage system. Hence state information would always be available.

Data Access and Storage

Tell architecture provides a SQL interface which can be utilized to query complex queries on relational data. The records in the database can be retrieved using data operations, like read and write, and then are stored in a key-value format.

In the data mapping, every relational record (or row) is stored as one key-value pair. Here the key is a unique identifier which are incremented monotonically and the value consists of a serialized set of all the versions of record. This type of row level scheme helps in minimizing the number of storage accesses. A more fine-grained storage scheme, like we can think of storing every version as a key-value pair would need additional requests for identifying the newly added versions. Although storing single versions always reduce network traffic, but it's better to opt for a technique that minimizes the network requests.

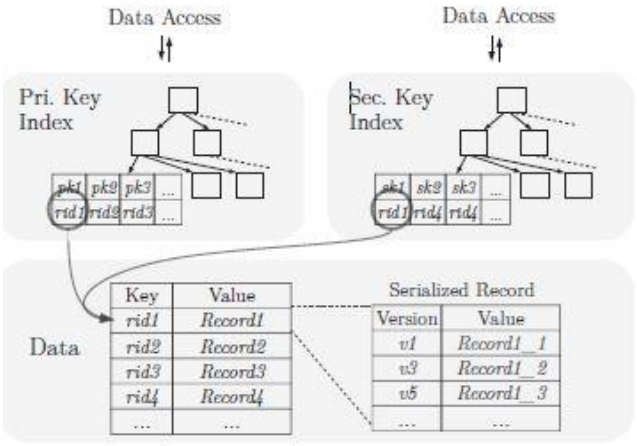

Fig 5. Shared-data Database

Processing Nodes (PN) are based on index structure for reading the records. Indexes contain the references to the row identifier (rid, as shown in figure above). Using index, a PN can retrieve the records with all of versions. The records are kept in transaction buffer and any update to the records are directly made to the newly added version. In the last, the record is applied at commit time.

OLTP execution can be parallelized and also scaled using number of PNs. For example, an OLAP query might want to execute an aggregation query which involves accessing all records of a table. For processing this query, a PN must read all the records of the table. An efficient lookup mechanism using indexes is shown below.

Latch-Free Index Structures (B+ Tree Index)

Indexes are an efficient way for lookup to identify the matching records for a given attribute (or a set of attributes). In a distributed environment, the absence of latches is good as network communication increases latch acquisition cost. Tell uses a latch-free B+ tree as its structure.

The B+ Tree index is completely stored and maintained in the storage layer. Every node is basically stored as a key-value pair and these can be accessed by all Processing nodes. The procedure can be explained as, on lookup a particular transaction accesses the root node of the tree and then traverses

until the leaf-level of the tree is reached. Here caching is also done to improve traversal speed and minimize storage system requests.

Garbage Collection in Shared data databases

Records Garbage collection is necessary to prevent data items/records from growing infinitely as Updation add new versions and over the time records become larger. The two garbage collection strategies described in this paper are as follows:

- Eager, here cleaning up of data records and indexes are the part of an update or read operation respectively.
- Lazy, it performs garbage collection in a background task that runs in a regular intervals of time. This is generally used for rarely accessed records.

Buffering Techniques in a Shared-data Architecture

The three approaches presented in the paper to improve buffering in a shared-data architecture are described as follows:

- **Transaction Buffering:** Here the transactions processed on a particular snapshot and therefore don't require the latest version of a record. Every transaction maintains a private buffer that caches all of the accessed records for the duration of that particular transaction's lifetime. The drawback is that there is no shared buffer across transactions.
- **Shared Record Buffering:** It is used by all the transactions on a processing node. This strategy is implemented using version number sets. Here by comparing the version number set of the transaction's snapshot descriptor which includes version number set of the buffered record and this allows us to check whether the buffer entry can be used or the transaction is too recent.
- **Shared Buffer with Ver. Set Synchronization:** This buffering scheme is a variant of shared record buffering strategy. The main idea here is to use the storage system for synchronizing the version sets of the records. The plus point here is that a Processing node can verify if a buffered record is valid by fetching its version number set.
    - This strategy helps from a workload with a high read ratio as cache units will be invalidated less often. The point is the higher the update ratio, the more buffered records are invalidated.

Experimental Evaluation

Implementation and Environment for the experiments

- Here all the processing nodes (PN) have synchronous processing model one thread process one transaction at a time
- PN interact with RAMCloud (RC) Storage system. RC is a strongly consistent in-memory key-value store designed to operate in low-latency networks
- Data is accessed using a client library that supports atomic get and put operations
- The backup mechanism synchronously replicates every put operation to the replicas and thereafter asynchronously writes it to persistent storage.

- The replication factor (RF) specifies the number of data copies. RC uses replication for fault-tolerance only.
- All requests to a particular partition are sent to the master copy.

Benchmarking environment considered

Parameter	Specification
Processor	12 servers each with two quad core Intel Xeon E5-2609 2.4 GHz 2 processors per server
RAM	128 GB DDR3
SSD	256 GB Samsung 840 Pro
Network	40 Gbit QDR InfiniBand

Table 1. Benchmarking environment details

TPC-C Benchmark used for the experiments

- An OLTP database benchmark.
- Every terminal operation results in one database transaction.
- Specifically, wait times are removed so that terminals continuously send requests to the PNs
- The size of the database is determined by the number of warehouses (WH). The default population for every TPC-C run is 200 WH's.
- This benchmark is write intensive with a write ratio of 35.84%
- **TpmC**: Throughput metric transaction rate is, the number of successfully executed new-order transactions per minute.
- The benchmark is executed 5 times for each configuration and the presented results are averaged over all runs. The measured performance was predictable and the variations were very low.
- For read intensive scenarios a TPC-C mix is considered. This mix consists of three transactions and provides a read-ratio of 95.11%

Experiments with scaling behavior of Tell

- Varied number of PN, SN and commit manager to check the scaling performance of tell
- It basically explains about the behavior and observation that TELL can scale with increasing number of nodes

Experiment on Processing Node

- **Write intensive**

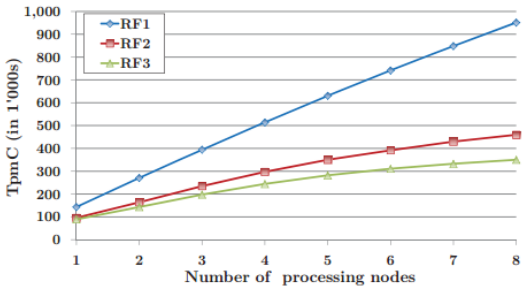

Fig 6. Scale out Processing (Write Intensive)

- Tell can scale with an increasing number of nodes.
- With no replication (RF1), throughput increases with the number of processing resources
- It suffers from data contention on the warehouse table and therefore throughput does not increase linearly. (grows from 2.91% (1 PN) to 14.72% (8 PNs))
- Increasing the replication factor adds overhead as updates are synchronously replicated.
- Synchronous replication increases latency and consequently affects the number of transactions a worker thread can process.
- Increasing data contention causes more aborts and throughput decreases
- **Conclusion:** Thus we can conclude that in write intensive task, as we increase the number of processing node the throughput increases in a case when the replication factor is set to 1. However, as we increase the replication factor overhead is added which degrades the throughput performance.

- **Read intensive**

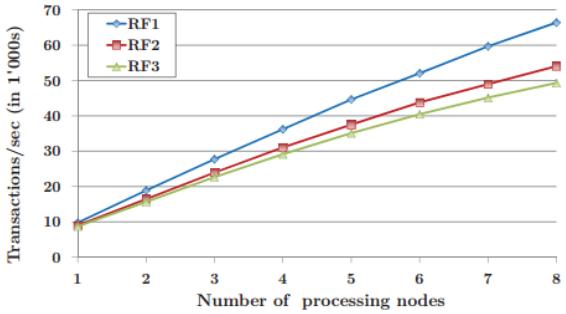

Fig 7. Scale out Processing (Read Intensive)

- Read operations only retrieve records from the master copy. Hence the latency is not affected by increase in replication factor.
- Higher the read-ratio of a workload, the less the performance impact of replication which is clearly seen in the figure where the performance is not affected majorly in terms of increase in workloads

- **Conclusion:** This emphasizes that Tell enables scalable OLTP processing with an increasing number of processing resources. The performance loss due to synchronous replication under write-intensive workloads is a common effect related to and is not specific to the shared-data architecture.

Experiment on Storage layer

- Figure shows the scale out of the storage layer.

- In all configurations the storage layer is not a bottleneck, and therefore, the throughput difference is minimal
- Figure explains about the behavior and observation that TELL can scale with increasing number of nodes
- The configuration done with number of storage nodes = 3 cannot be executed on a setup where the storage node is greater than 5.
- **Conclusion**: The number of storage nodes in the cluster should be determined based on the memory requirement and not be dependent on the CPU power.

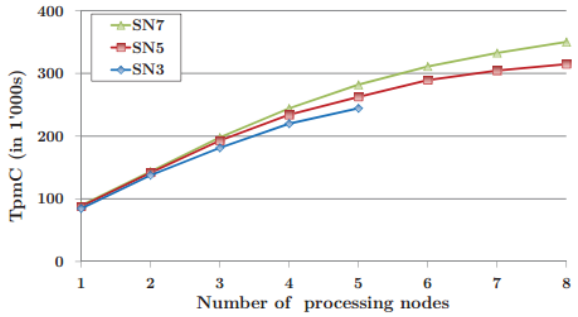

Fig 8. Scale out Storage (Write Intensive)

Experiment on Commit Manager

- Commit Manager are mainly used to assign TID's and keep track of completed transactions. They do not generally perform complex operations.
- TPC-C benchmark with a synchronization delay of 1ms caused no significant impact on throughput and on the transaction abort rate.
- **Conclusion**: Thus with a delay there is no impact on the throughput generated by the commit manager. Hence this is not considered as a bottleneck.

Commit Managers	**1**	**2**	**3**
TpmC	958'187	955'759	955'608
Tx abort rate (%)	14.75	15.59	15.91

Table 2. Commit Manage (Write Intensive)

Comparison of Tell with Partitioned Databases (VoltDB and MySQL cluster)

- **VoltDB: -** VoltDB is an in-memory relational database that partitions the data and then serially executes transactions on each of the partition. The more transactions can be processed by single partitions, the better VoltDB scales.
- **MySQL: -** MySQL Cluster is also a partitioned database with an in-memory storage engine. Here, a cluster configuration consists of 3 components which are as follows:
    - **Management nodes (MN)** that monitor the cluster
    - **Data nodes (DN)** that store data in-memory and process queries, and
    - **SQL nodes** that provide an interface to applications and act as federators towards the DNs.

- **Experiment 1:** Here the figure presents the peak TpmC values for the TPC-C standard mix with RF3 and the no. of CPU cores are varied here.

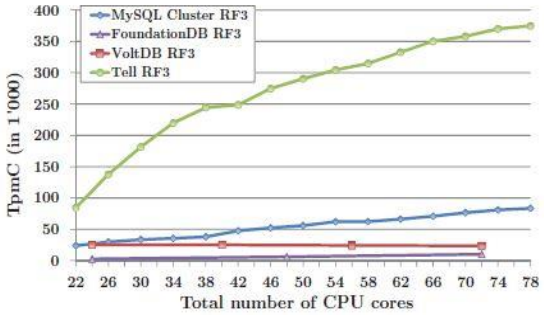

Figure 9: Throughput (TPC-C Standard)

  - It can be seen from above figure, a minimal VoltDB cluster consists of 3 nodes (i.e., 24 cores). However, the minimal Tell configuration with 22 cores consists of 1 PN (4 cores), 3 SNs (12 cores), 2 commit managers (4 cores), and 1 MN (2 cores).
  - When compared with Tell architecture, VoltDB and MySQL clusters do not scale with the number of cores. It can be observed from figure that Tell reaches a throughput of 374,894 TpmC with 78 cores, MySQL Cluster and VoltDB achieve 83,524 and 23,183 TpmC respectively.
  - In particular, VoltDB suffers from cross-partition transactions as throughput decreases the more nodes are added. It can be observed by VoltDB's high latency as shown in table below. MySQL Cluster is slightly faster than VoltDB because single-partition transactions are not blocked by distributed transactions.
  - **Conclusion:** The shared-data architecture enables much higher performance than partitioned databases for OLTP workloads that are not fully shardable.

		Small 22-24cores (mean $\pm$ σ, ms)	Large 70-72cores (mean $\pm$ σ, ms)
Standard	Tell	14 $\pm$ 17	32 $\pm$ 41
	MySQL Cluster	34 $\pm$ 26	43 $\pm$ 40
	VoltDB	706 $\pm$ 2159	655 $\pm$ 1875
	FoundationDB	149 $\pm$ 183	120 $\pm$ 138
Shard	Tell	14 $\pm$ 17	32 $\pm$ 41
	VoltDB	62 $\pm$ 102	22 $\pm$ 59

Table 3: TPC-C transaction response time

- **Experiment 2:** In the second experiment, Tell has been evaluated for a workload that is optimized for partitioned databases. To that end, TPC-C benchmark has been modified and all cross partition transactions has been removed. The Figure below compares the Tell to VoltDB and MySQL Cluster using the TPC-C shardable with RF1 and RF3.

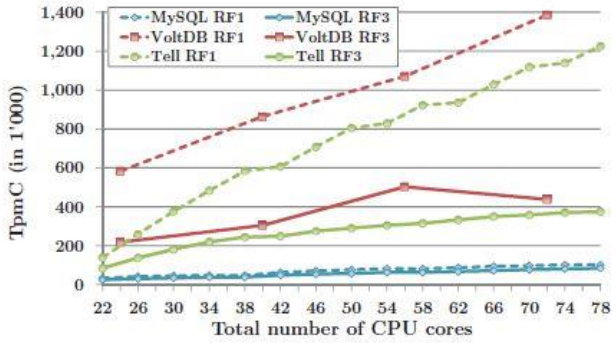

Figure 10: Throughput (TPC-C Shardable)

- Here the number of cores has been varied. With this given workload, VoltDB fulfills the scalability promises and achieves more throughput when compared with Tell.
- Talking about the peak performance of VoltDB with RF1 is 1.387 Mio TpmC. However, Tell achieves 1.225 Mio TpmC, which is in all 11.7% less than VoltDB.
- It has been observed that MySQL cluster is only 1-2% faster than with the standard workload.
- **Conclusion:** The results shows that the shared-data architecture (as in Tell) enables competitive OLTP performance. Even with a perfectly shardable workload, the achieved throughput is in the same ballpark as state-of-the-art partitioned databases.

Comparison to Shared-Data Databases (FoundationDB)

- Here FoundationDB, another shared data database has been compared with Tell architecture.
- FoundationDB is known for its fast key-value store and has recently released a “SQL Layer” that allows for SQL transactions on top of the key-value store.
- The environment (i.e., TPC-C Benchmark) runs FoundationDB 3.0.6 with in-memory data storage and RF3 (redundancy mode triple).
- The smallest configuration (24 cores) achieves 2,706 TpmC and the largest (72 cores) reaches 10,047 TpmC.
- We can see from these results that Although FoundationDB scales with the number of cores, the throughput is more than a factor 30 lower than Tell.
- **Conclusion:** As the SQL Layer is still new and it can be believed that FoundationDB will work on improving the performance of the SQL Layer in the near future. Nevertheless, these results indicate that if not done right, shared-data systems show very poor performance.

Network

- This experiment describes the importance of low latency data access in a shared-data architecture. The experiment compares throughput performance of 10GB Ethernet with InfiniBand network
- The network is used to communicate between PN - SN as well as amongst SN's. The RF is 1 and SN is 7 for the experiment.
- As per the figure, the TpmC results on InfiniBand Network at 6times greater than on Ethernet Network irrespective of the PN used. The difference lies behind the network latencies.

- InfiniBand Network allows data to be accessed faster while the processing time decreases. This is because of the fact that they bypass the network stack of OS thus providing low n/w latencies.
- TELL being a synchronous Processing model (Transactions blocked until data arrived), the reduction in latency has positive effect on throughput.

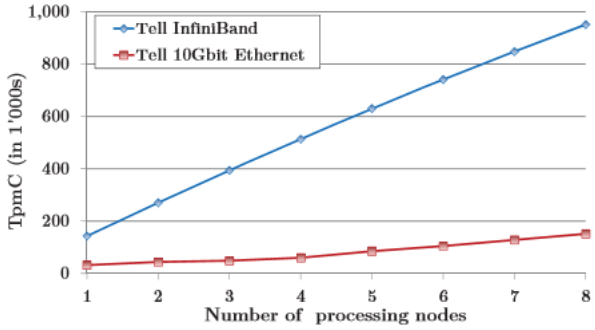

Figure 10: Network (write intensive)

	TpmC	Latency (mean ± σ, ms)	TP99 (ms)	TP999 (ms)
InfiniBand	958,187	10.69 ± 13.02	76.48	100.62
10Gb Ethernet	151,632	69.41 ± 87.99	542.03	644.15

Table 4: Network Latency (write intensive)

- Explanation of Network latency results:
  - Column 1 - results of fastest configuration with 8 PN
  - Column 2 - mean transaction response time and standard deviation
  - Column 3 - 99th percentile response time.
  - Column 4 - 99.9th percentile response time.
  - The low number of outliers indicates that both networks are not congested

Buffering Strategies

- The below shown figure shows represents the comparison of the buffering strategies presented earlier in this report. The experiment configuration uses 7 SNs and RF1.

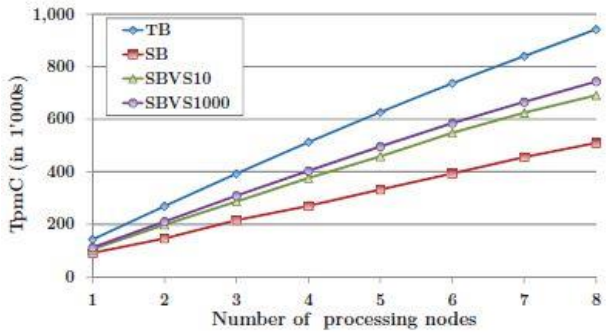

Figure 11: Buffering Strategies (write-intensive)

- The transaction buffer (shown as TB) used in all of the previous experiments is found to be the best strategy for the TPC-C as it reaches the highest throughput.

- However, the shared record buffer (shown as SB) performs worse because here the overhead of buffer management outweighs the caching benefits.
- The shared buffer with version set synchronization (shown as SBVS10 and SBVS1000) tested with cache unit sizes of 10 and 1000 had a considerably higher cache hit ratio.
- **Conclusion:** The transaction buffer strategy is found to be best when compared from throughput chart shown above.

Application

The shared database pattern is a useful when there is a need to have the same data instantly available to all consumers. Basically the data should be available and the recovery of the data should be done as quickly as possible.

This pattern can be used in various real time scenarios where important data can be affected and the replication of the data across instances would keep the data available throughout the life cycle of the system.

Conclusion

In this report we describe TELL as a database system based on a shared-data architecture. TELL decouples Transaction Query Processing into two layers to enable elasticity and workload flexibility. The distributed record manager is used to store data and shared among all database instances.

The report compares Tell to alternative distributed databases. It was noticed that TELL achieved a higher throughput than VoltDB, MySQL Cluster, and FoundationDB in the popular TPC-C benchmark. For the experiments the shared database was scaled with different number of cores throughout its evaluation.

With the above benchmarking observations and evaluation results, it can be concluded that the shared data architecture enables high performance irrespective of the workload assumptions thus being elastic.

Criticism

- Maintenance and development costs can increase: Development is harder if an application needs to use database structures which aren't suited for the task at hand but have to be used as they are already present.
- Administration becomes harder: Administration of all the instances become difficult which is not addressed in the paper.
- Upgrading: Upgrading the each of the shared instances in time consuming and an issue with the shared data architecture. This is not addressed in the paper.
- Redundant data due to replication: Since the same data is replicated across various nodes redundant data will be present. The paper does not discuss about issues with redundancy.
- Security issues across shared nodes are not discussed in this paper.
- If the data is changed then making the changes across all the nodes adds to an overhead and is more time consuming.

References:

- J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. L´eon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: providing scalable, highly available storage for interactive services. CIDR'11, pages 223–234, 2011.
- M. Cahill, U. R¨ohm, and A. Fekete. Serializable isolation for snapshot databases. SIGMOD'08, pages 729–738, 2008
- P. Bernstein and N. Goodman. Concurrency control in distributed database systems. ACM Comput. Surv., 13(2):185–221, 1981.