Scalable OLTP in the Cloud: What's the BIG DEAL?

The Database AND the Application Have a BIG DEAL: Their Isolation Semantics

Pat Helland
phelland@salesforce.com
Salesforce
San Francisco, California, USA

ABSTRACT

The pursuit of scalable OLTP systems has been the holy grail of my career. Because OLTP systems are typically split into applications and databases, the isolation semantics provided by the DB and used by the app have a major impact on the scalability of the OLTP system as a whole. The isolation semantics are a BIG DEAL!

This thought experiment explores the asymptotic limits to scale for OLTP systems. An OLTP (OnLine Transaction Processing) system is a domain-specific application using a RCSI (READ COMMITTED SNAPSHOT ISOLATION) SQL database to provide transactions across many concurrent users. This interface provides the contractual BIG DEAL between OLTP databases and OLTP applications.

Focusing on the BIG DEAL, shows today's popular databases unnecessarily limit scale. Similarly, we identify common app patterns that inhibit scale. We can reimagine the way we build both databases and applications to empower scale. All while complying with the established SQL and RCSI interface (i.e., the BIG DEAL).

Perhaps, this can provoke discussions within the database community leading to new opportunities for OLTP systems. To me, that would be a big deal! .

1 INTRODUCTION

Good academic papers make you think. Great academic papers cause weeks of contemplation. The CIDR 2023 paper "Is Scalable OLTP in the Cloud a Solved Problem?"[38] prompted a personal reflection on the essence of OLTP[5, 15] systems, their databases, their applications, and how they scale.

What are the limits to scale given unbounded resources? What is a perfectly scalable application? With such an application, can a database avoid adding its own limits to scale?

1.1 A Thought Experiment about Scale

This Gedanken Experiment[36] explores a narrow question:

"What are the asymptotic limits to scale for cloud OLTP systems?"

To consider this, we first define the parts of the question!

1.2 What's OLTP and What's a Transaction?

OLTP (Online Transaction Processing) systems provide domain specific business applications (e.g., banking, retail, travel booking) for humans interacting with computers. Low latency (10s to 100s of milliseconds) and high availability (typically targeting 99.99% availability) are important. In OLTP, both the app and the DB manage many concurrent small units-of-work. **App-units-of-work** happen as humans and/or other computers interact at a rapid rate. They examine database data, write new records, and send messages back. **DB-units-of-work** are database transactions (TXs).

Each app-unit-of-work comprises one or more DB-units-of-work hopefully fast enough for human users.

Figure 1: The BIG DEAL is the semantic contract provided by the DB and used by the App. An OLTP application provides domain specific behavior to online users.

Start with Existing SQL Database APIs → The BIG DEAL
An OLTP system comprises an online application running on top of a database. To make the thought experiment more challenging, the application programming interface to the database is assumed to be compatible with today's SQL relational databases. Specifically, we consider the very common RCSI (Read Committed Snapshot Isolation)[1]. In this paper, we call it the BIG DEAL[2]. See Figure 1.

Many scalable systems proposals evolve the semantics of the underlying DB or key/value storage system. Here, we explore scalability without renegotiating the contract defined by the BIG DEAL.

What is meant by "Asymptotic Limits to Scale"? Our thought experiment assumes unlimited compute, storage, and network resources. *When does adding more resources fail to increase scale?*

Asymptotic computational complexity ignores smaller deployments, focusing on the characteristics of a system as its size approaches infinity. Specifically, this is not a paper about performance of some solution as long as the OLTP system is fast enough for human users. Resource consumption costs are not a concern[3].

1.3 Conclusion #1: BIG DEAL Semantics Scale

Today's SQL RCSI interface supports arbitrary scale assuming the database and the domain specific application both do their part.

The BIG DEAL splits scaling responsibilities:

- **Scalable apps** don't concurrently update the same key.
- **Scalable DBs** don't coordinate across disjoint TXs updating different keys.

 14th Annual Conference on Innovative Data Systems Research (CIDR '24). January 14-17, 2024, Chaminade, USA

[1] Our usage of RCSI is called READ CONSISTENCY in Oracle[28] and READ COMMITTED in SQL Server[32] and PostgreSQL[29]. Snapshot Isolation is the default in Oracle, PostgreSQL, and SQL Server Azure. , SQL Server supports snapshot isolation by setting READ_COMMITTED_SNAPSHOT_ISOLATION=ON.
We assume READ COMMITTED with SNAPSHOT ISOLATION.

[2] "BIG DEAL" connotes BOTH a contractual relationship AND a thing of importance.

[3] Linear increases in cost – O(N) scaling – would be ideal but scalable solutions typically include a small O(N-log-N) increase in addition to the O(N) work.

The BIG DEAL (SQL + RCSI) Semantics Empower Scalable DBs and Apps

Today's DBs Coordinate across Disjoint Transactions
→ ***That Doesn't Scale!***

Today's Apps Too Often Update the Same Key Concurrently
→ ***That Doesn't Scale!***

1.4 Conclusion #2: The BIG DEAL Offers Guarantees & Exposes Weaknesses

The SQL RCSI interface has two aspects seen by databases:

- **Guarantees** that the DB must provide to the app.
- **Weaknesses** that today's RCSI apps must tolerate or they are buggy. These weaknesses may be exploited by the DB.

The BIG DEAL provides guarantees from the DB to the App:

- **Reads:** A scalable application can read all it wants.
- **Updates** to disjoint records don't coordinate across TXs.
- **Row-locks** on disjoint records don't coordinate across TXs. See §2.2.

Applications must tolerate these BIG DEAL weaknesses today:

- **Reads return snapshots:** Records have no "current" value.
  - *Snapshots are immutable* - Stale data is accurate!
- **There is no NOW in a BIG DEAL database!**
  - *There is no global NOW in an RCSI database!*
  - *Time increases subjectively for each observer within the DB.*
  - *Time provides ordering of BEFORE & AFTER across record-versions, snapshots, transactions, servers, and the data visible outside the DB.*
- **Transactions may abort any time** but not too often.
- **SELECT with SKIP LOCKED** may subset the set of qualifying records as it returns results.

1.5 Conclusion #3) Domain Specific Apps Change Behavior as They Scale

Scalable apps use the BIG DEAL to provide domain specific function:

- **Guarantees from the DB** are used to build scalable apps.
- **Weakened domain specific behavior** is seen by human users as the application scales.

Given sufficient traffic to an OLTP application, concurrent updates or row-locks to the same records inhibit throughput.

To scale, an application must weaken the behavior exposed to its users. Business state can't be synchronously updated by multiple users. Recent changes to shared business state take time to accumulate. They gain clarity over time.

Applications change business behavior as they scale:

- *Business state after recent changes becomes less specific.*
- *Commitments to future business work becomes less specific.*

Apps introduce ambiguity in biz domain specific ways:

- Online retail only promises to "Usually ships in 24 hours"
- Finances of a large company may take days to summarize.

1.6 Conclusion #4) Scalable Databases Have No "Current" Data, Only Snapshots

With the BIG DEAL, applications only read data previously committed. Everything they read is frozen in time as-of their snapshot time. There is no notion in the BIG DEAL of reading the "current" value of data, only past values.

Each record update creates a new record-version visible to later snapshots. Before an update is committed, the DB ensures that no competing updates have changed that record. Future updates are protected from concurrent changes.

Everything in the database is versioned by the record-version commit time. This is necessary to provide snapshot reads for the BIG DEAL. *Organizing data by its creation time* empowers scaling:

- **Reads** ← ***old*** record-versions as-of a past snapshot.
- **Row-locks** ↔ ***Locked records unchanged*** until commit.
- **Updates** → ***new*** record-versions for later snapshots.

Today's Databases Don't Scale!

Reads & writes fight to access the "current" value of a record.

1.7 This Paper Hopes to Provoke Debate

This paper hopes to reflect on long held assumptions and think systematically about the essential characteristics of scalable OLTP systems. How can scalability happen and what are we doing wrong as we tackle the problem?

How do databases get scaling wrong? For decades, I assumed each record had a "home" for its "current" value within a partition, a block, or a data structure (e.g. B+Tree[8]). I assumed the DB had to go to the record's "home" to update and/or read the "current" value. For more than 45 years, I didn't question this assumption. I repent[4] that belief. The BIG DEAL reads the past and verifies to avoid conflicts in the future. There's no NOW, no "current" values, and no "home for a current value".

How do applications get scaling wrong? Most OLTP applications synchronously aggregate application knowledge as it changes. As systems scale, modifying records shared across users becomes untenable due to increased contention. If an application cannot aggregate changes synchronously, the best it can do is aggregate knowledge gradually in later transactions. Scalable applications coalesce their knowledge of the past slowly. They are also more ambiguous about their commitments for the future. See §3.

We should proactively evaluate performance: Over many years, scale and performance work has usually seemed a lot like whack-a-mole[5]. This paper aspires to stand back and reflect on the essence of scale and its limitations in an OLTP system. Hopefully, it inspires others in the community to reflect on this, too.

This paper hopes to "pay it forward" by provoking debate about:
"The Limits to Scale of OLTP Systems"

1.8 What's Coming in the Rest of the Paper?

Upcoming sections cover the following topics:

- The BIG DEAL – What Does the DB Provide to the App?
- Evolving Applications to Increase Scale
- The "Key" Scaling Challenges in Today's MVCC Databases
- Rethinking OLTP Databases: A Strawman Architecture
- Scalable Application Queuing
- Massive Scale: It's about Time!
- Related Work, Acknowledgements, and Conclusion

[4] **Repent:** "To think of (an action or omission) with deep regret or remorse."

[5] **Whack-a-mole:** (idiomatic, chiefly US & Canada) The practice of trying to stop problems, etc., that repeatedly occur in an apparently random manner; also, the act of dealing with such matters in a piecemeal way without achieving a complete solution.

2 THE BIG DEAL: WHAT THE DATABASE PROVIDES TO THE APPLICATION

This section provides a perspective on the BIG DEAL between the app and the DB. What are the abstractions and what are the tricks that empower scale? This section describes the guarantees from the database to the application. Later sections will describe how to exploit the weaknesses not guaranteed to the application.

2.1 Historic Perspective: Isolation Semantics Have Relaxed to Increase Scale

Prior to the advent of relational systems, locking and isolation took many forms[5, 15–17]. Systems such as IMS[21, 22] provided non-relational access to data with different isolation modes. IMS Fast-Path had special mechanisms to allow highly concurrent updates to in-memory values[12].

Defined in the mid-1970s, *serializability* [4, 5, 11, 15] gave a notion of perfect isolation across concurrent TXs as the perception of a serial order of execution. Supporting SERIALIZABLE isolation caused scaling limitations that soon became important concerns.

Over almost five decades, SQL databases vendors provided options to their apps allowing relaxed isolation guarantees in exchange for increased scale. The BIG DEAL between the app and DB evolved! While there are many varieties of isolation, we focus on three major steps forward (on scale) and backward (in isolation):

- **SERIALIZABLE → REPEATABLE READ:** This relaxed the promise that missing records remain missing. While the actual records read remain stable, the absence of records is not ensured to the reading transaction.
- **REPEATABLE READ → READ COMMITTED:** This relaxed the promise that records seen by a transaction don't change before the transaction commits. Transactions may be aware of concurrent work when a record is read more than once. Readers may stall if they attempt to read a record concurrently being written by another transaction.
- **READ COMMITTED → RCSI:** RCSI (READ COMMITTED SNAPSHOT ISOLATED) allows readers to ignore concurrent writers. Reads (i.e., SELECTs) return records committed before their snapshot time. The database must keep multiple versions of updated records[6]. This is called MVCC (Multi Version Concurrency Control)[4, 5].
- **RCSI → RCSI (with SKIP LOCKED):** Using SKIP LOCKED allows the database to subset the results returned from a SELECT. Just leaving out data is OK[7]. See §2.3.

2.2 Row-locks: Separating Application Semantics from Scaling and Concurrency

Most existing applications depend on row-locks for their correctness. This works well as long as they don't scale too large[8].

Row-locks perform two BIG DEAL functions. They provide correctness guarantees to the application and allow the application to ask the database for help with concurrency across transactions.

Row-locks – Probabilistic concurrency control: Since transactions may abort without cause, when locks are granted (for concurrency control), the owning transaction may abort. Hence, the order of work derived from lock ownership is not a guarantee[9]. Using row-locks to provide application order of execution is very useful, even if the order is occasionally violated.

Row-locks – Guarantee application semantics: Row-locks offer important semantics to the application and its locking transaction. Row-locks ensure that transactional changes can depend on the values seen within the locked records. Updates to other records must not commit unless the locked record remains unperturbed.

2.3 SKIP LOCKED to Expand Concurrency

Even with RCSI, some application patterns are challenging. An application may want to avoid waiting for a lock to increase concurrency. One option is to use SELECT NOWAIT to return an error when a SELECTed record is locked. Related to this is SKIP LOCKED[10]. The SQL query:

SELECT <query> FOR [SHARE or UPDATE], SKIP LOCKED

allows the DB to omit locked records from the result. When unlocked, these records may appear in later SELECT statements.

2.4 Access Patterns for RCSI: The Sweet Scaling of Snapshot Isolation!

Let's consider the access operations performed by a DB in four broad categories: read, row-locks, updates, and transaction commit.

Read is always as-of a snapshot: Only record values committed before the snapshot are included in the result set. These reads may be based on an exact key or a key-range.

Row-locks guarantee "What you see is what you get." The locking transaction knows records won't be changed by other transactions. Competing transactions usually wait to allow the lock holder to go first but that may be flawed.

Update combines both snapshot time and commit time: Each update is performed on a single key as-of a snapshot time. If another TX commits an update to the same key after that snapshot time, this update won't survive. Statement restarts may (or may not) preserve earlier work within the TX, repeating the update with a later snapshot including the interloping change.

Transaction commit is probabilistic: Commit may occur if the RCSI rules are enforced. Records returned by SELECT FOR SHARE or SELECT FOR UPDATE may not have been changed by another transaction since the snapshot time of the SELECT statement. Records updated by the transaction cannot have been changed by another transaction since the snapshot time of the update. Commit is only guaranteed to survive after it has completed. Beforehand, it's a gamble.

[6] Snapshot isolation proved very popular with application developers and RCSI is available within Oracle, PostgreSQL, and SQL Server amongst others[28, 29, 32]. Read-only transactions do not semantically stall waiting for updating transactions.

[7] SKIP LOCKED is valuable for high-traffic application queues. See §6 and §6.3.

[8] Row-locks are moot when scalable apps avoid concurrent updates to the same records. This paper considers a BIG DEAL DB for both scalable and non-scalable apps.

[9] Many distributed systems algorithms differentiate between liveness and safety. Safety means the algorithm doesn't do the wrong thing. Liveness means it avoids inordinate delays. Similarly, row locks within RCSI have two aspects. They offer a guarantee of safety to the locking transaction. Specifically, the transaction will commit only if the value locked has not been changed by a concurrent transaction. Row-locks also help the liveness of transactions when the DB uses them to function as a traffic cop.

[10] Recent versions of Oracle, MySQL, and PostgreSQL call this SKIP LOCKED. SQL Server's option is READPAST. DB2's variant is SKIP LOCKED DATA (DB2 has READ COMMITTED without MVCC).

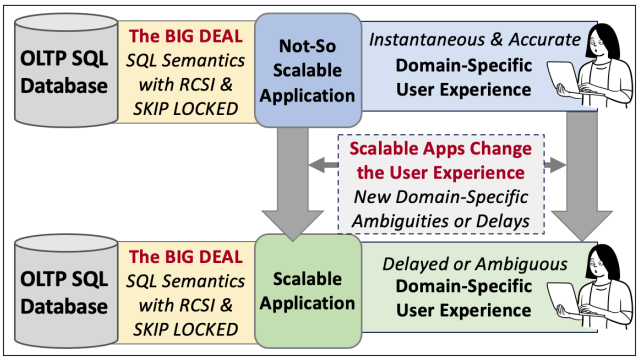

Figure 2: As applications evolve for increased scale, users see fuzzier promises about the future. Past status is more ambigous at first and crisper as time passes.

3 EVOLVING APPS TO INCREASE SCALE

Evolving an app for scale usually requires changing its behavior. Each app performs a role in some business domain such as online retail, ERP (Enterprise Resource Planning), banking, hotel or airline reservations, logistics, service call centers, and more. See Figure 2.

Simple non-scalable applications can immediately expose crisp, clear, and accurate business state. As domain-specific apps are augmented for scale, many of these details must become ambiguous.

Of course, crisp, clear, and accurate knowledge within the app may or may not reflect the real world. Apps managing tangible goods like hotel rooms or inventory may have inaccurate knowledge of the physical world. To cope with this, even not-so-scalable applications sometimes present fuzzy and/or delayed information.

Hot-spot records impede application scale. To reduce contention on hot-spots, domain specific tricks are employed. Knowing a "pretty good" estimate of the recent inventory for a book, along with its expected purchase rate, allows the application to promise the book "Usually ships in 24 hours."

Users of the scalable OLTP system frequently can't tell the difference between *ambiguity within the computer* and *the computer's ambiguous understanding of the real world.*

These ambiguities are domain-specific & exploiting them must be domain-specific. This exploitation cannot easily be automated by a general purpose database or even an application execution platform that is not domain specific.

3.1 My Application Is Successful... Help Me!

How do apps experience scaling challenges? These only emerge as the usage of the application increases. Both business semantics and the management of the application itself can impede scale.

Apps are rarely designed for scale when first created: With rare exceptions, the first version of an application focuses on functionality and ease of use. Simple techniques are used to manage shared business data by directly updating shared records within each app-unit-of-work. Typically, business state is aggregated synchronously as changes occur. This is a wise choice. Most applications never need so much scale that simple synchronous aggregation is a problem. Working hard to ensure scaling before launching the application only adds risk.

Simple strategies work well until the app is successful. As the pace of work rises, concurrent writes to the same key throttle access; app throughput slows. Work queues get clogged tracking their head and tail. Writing the inventory balance throttles warehouse shipments. Adding more servers doesn't increase scale.

Synchronous aggregation updates business values as app-units-of-work are performed. Consider an app managing warehouse inventory. As products ship, the item count is subtracted from the warehouse's inventory. Incoming supplies are added to the inventory. When the transaction traffic rate is low, this is simple and effective. As traffic increases, this may throttle business[11].

Cross-TX contention on DB records may come from internal app maintenance. Sophisticated apps interleave synchronous online work with asynchronous "batch style" processing. For example, logically deleted records may be marked as not visible to ongoing work. The application may later copy these "deleted" records to an archival storage in batches driven by an application queue. Even later, they may be garbage collected in batches.

Application queues may impede scale. Application queues are commonly used for asynchronous application tasks. These may interleave many small pieces of work or to ensure some long running multi-step workflow completes. Application queues are easy at a small scale. They can be implemented by keeping work items as records in a table. Combine the table with a head record and a tail record to get a queue. This works great until it doesn't. Under extreme scale, application queues are complex. See §6.

3.2 Logarithmic Rollup of Business State

If business traffic grows exponentially, aggregating business state is very hard. Aggregating in stages over multiple DB transactions can address this. A sophisticated application can partition its changes both by time and by other dimensions such as region, territory, management structure. For each dimension, it aggregates by time windows. Successive stages can break the aggregation into manageable pieces gaining better resolution by iterating in stages.

Example logarithmic rollup – Large political elections: Large political elections are great examples of scalable, distributed, and parallel processing. In the United States, a national election comprises tallying up results from many states, many counties per state, and many voting precincts per county. Results gradually accumulate after ballots close throughout these geographies.

Accuracy fights with timeliness: In some elections, the probable outcome is clear very early. Trends and statistics based on exit polls or early ballot counts are rapidly reported. A pattern favoring one outcome may cause rapid acceptance of the probable winner. Other times, the race is very close. Votes are counted and recounted. The election outcome may be unknown for days, weeks, or longer.

Refining accuracy is hierarchical & logarithmic: Total votes are aggregated across the geographic regions in a hierarchy. Recounted vote totals flow from precinct to county, state, and national totals. Each recounted total ripples up the hierarchy step by step.

A new tally at a single precinct is gradually rolled into its county's total. Changes to the county total stimulate later changes to the state total, etc. In this example, the depth of the political hierarchy is four and each new tally causes three asynchronous refinements.

This becomes a logarithmic roll up of the summary count bounding the churn on stages up the hierarchy. See Figure 3.

[11]Long considered important to OLTP work, the TPC (Transaction Processing Council)[34] benchmarks forced synchronous aggregation. For example, TPC-B (1990) was a "debit-credit" benchmark requiring atomic updates to the "branch balance" for each teller's bank interaction.

Chunking time to reduce churn: In the chaotic minutes after election polls close, there's a flurry of activity. At each stage of the hierarchy, new summary counts stall a bit to incorporate changes from below. This allows consolidation of results allowing scale.

Rapid churn leads to overlapping chunks of time: At large scale, earlier time chunks may not yet finish consolidation before new time chunks start. For example, the election results sent 10 minutes ago from a precinct to the county may not yet be incorporated when the next update from that precinct is sent to the county. Many overlapping time chunks may be chasing each other as they roll together. Newer chunks rolling up recent changes are blurry and older chunks more accurate. Last week's activity is more accurately summarized than this week's activity.

3.3 Gaining Wisdom and Clarity with Age

For large applications, the past is clear and the present is blurry.

The past may be examined in many dimensions. Time is analyzed hierarchically by years, quarters, months, weeks, days, and more. Quarterly financial results include errata to previous financial summaries. Business transactions are scrutinized by geographies, product types, business value, cost of goods and more. Today, it is common to dynamically modify the analysis of the business. Like viewing a digital photograph over the Internet, the picture starts out grainy and blurry but comes into focus over time.

Making commitments based on blurry knowledge: We've all seen the message: "Usually ships in 24 hours." This guarantees nothing but it's very useful. At scale, you want a good guess. Even if an airplane seat is reserved, the flight may be canceled; only when the plane door closes do we know who is on board. Stale knowledge about the old inventory state combines with the expected consumption rate to decide on "promising" a future shipment.

Closing the loop on good guesses: At scale, resources are promised if they'll probably be there. Big apps use workflows to confirm their promises have been met. After seconds or minutes, a confirmation email is sent. Given some time, blurriness clears up.

3.4 Premature Aggregation in OLTP Apps

Business aggregations (e.g., warehouse inventory) can be derived from the online changes from many domain-specific OLTP app-units-of-work. Each of these transactions remembers what happened (e.g., performing a shipment). Business aggregations can be adjusted later based on the "memories" from the online transactions.

Synchronous aggregation is challenging. Many OLTP apps aggregate values synchronously as they interact with humans. Public TPC benchmarks[34] (e.g., TPC-A, TPC-B, and TPC-C) mandated these synchronous aggregations. By slowly aggregating these business values, the application can scale in a domain-specific manner.

4 THE "KEY" SCALING CHALLENGES IN TODAY'S MVCC DATABASES

When the app AND the DB both scale, the system scales. Scalable apps avoid concurrent writes to any record. Scalable DBs avoid coordination across transactions performing snapshot reads. They also must avoid coordination across transactions updating or locking disjoint record keys. Unfortunately, today's database implementations fall short of these scaling requirements.

4.1 Snapshot Isolation: MVCC Helps Apps Scale (at Least in Theory)

Snapshots relax dependencies between reads and writes of DB records, increasing concurrency and scale. Theoretically, this means that reading committed versions of records as of a snapshot should not slow down when new updates to those records are performed.

MVCC databases maintain multiple versions of records. Today's major commercial databases do provide access to multiple versions of records. Unfortunately, they slow down access as they coordinate snapshot readers with concurrent transactions writing new versions of records. This impedes scale for OLTP systems.

Minimizing unneeded coordination across concurrent TXs. When concurrent TXs write to the same key, coordination across these transactions is semantically required by the BIG DEAL. When reading a key's snapshot old value, coordination with any other TXs is not required semantically. Scalable databases minimize or eliminate unnecessary coordination.

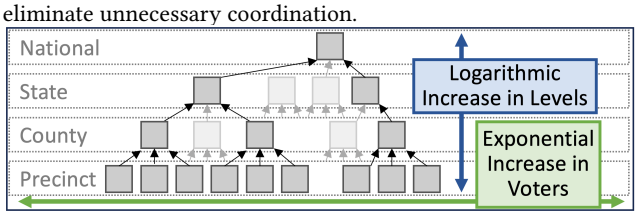

Figure 3: Business summaries roll UP hierarchically over time. ***Logarithmic roll up → exponential capacity for changes.***

4.2 Today's Implementations of MVCC: Fighting Your Way Home

Today's major DBs implement MVCC by moving older versions elsewhere. The current version has a "home" location holding the most recently committed version of the record or perhaps an uncommitted version. A record's "home" may be a partition, server, data structure (e.g., B+Tree), and/or block.

Updating a record happens by first moving any older versions of the record elsewhere and recording where it may be found. Then, the new version for the committed (or pending) new version of the record is stored at the home.

To read a snapshot version of the record, the reading transaction looks at the "home" for the record. If the version is later than the snapshot, the read climbs back in history consulting consecutively older locations until the latest version older than the snapshot is located. Today's major commercial databases all follow this pattern albeit with different strategies for where to place the older versions in their MVCC storage. See Table 1.

To update a record, exclusive access to the record's home is required. This causes infighting, contention, and coordination between the updating TX and any concurrent reading TXs.
Multiple record's may also experience cross-key contention when their "homes" are stored in common data structures.
This severely limits the scalability of the database and application(s) using it.

Coordination is needed to access the latest version: Accessing the "home" for the latest version requires a combination of shared and exclusive locking (or latching in DB parlance). Since a record's "home" may have multiple granularities, each must be

DB Product	MVCC Approach	Where's the Newer Record?	Where's the Older Record?	What Causes Coordination across Records?
Oracle	Undo blocks	Replaces older version	Special undo blocks	***Update-in-place*** at the older version's location
SQL Server	Temp table	Replaces older version	Stored in temp table	***Update-in-place*** at the older version's location
PostgreSQL	Version range kept with record key	Next to older record	Next to newer record	***Update-in-place*** next to the older version's location

Table 1: Today's popular MVCC implementations coordinate snapshot reads with concurrent updates. Storing new versions updates the older version's location requiring exclusive access and interrupting snapshot reads.

navigated. Shared access to the home partition, home server, home data structure (e.g., B+Tree), and/or a home page may be required.

Coordination is needed to access earlier versions. Since these implementations force MVCC readers to start out looking at the latest version of a key first, readers of snapshot values coordinate for the record's "home". From the home, locations of the older versions may be accessed. In many implementations these older locations may require coordinated access, too. Readers coordinate with concurrent writers and readers of each key. This happens even when reading snapshots that are semantically immutable!

Coordination is needed to access neighboring records. MVCC writers may need to coordinate access to key-ranges, even for neighboring keys. Accessing key-ranges in B+Trees or similar data structures that may be changing needs cross-transaction coordination. Readers coordinate with writers. Writers coordinate with readers. Readers coordinate with other readers! It's unfortunate!

Today's MVCC implementations coordinate across TXs.
This limits scale over and above the BIG DEAL's semantics.

4.3 Today's Partitioned (or Sharded) Databases: Repartitioning Online Is Very Difficult!

A similar situation occurs with partitioned databases. Each record is assigned a partition, typically by key-range. In today's systems, that is where the record is read and written[12].

When each record's changes go to ONE log, it has a "home". Partitioned DBs typically assign a record to a *home partition* based on its key. The partition creates new versions of the record by logging into the partition's transaction log. Each record (with a unique key) sees its new record-versions written to a single log. Atomic TXs across partitions need some variant of two-phase commit[5, 15, 26].

Write skew can skew-up your day! One major challenge associated with today's partitioned databases happens when the write traffic to the database does not match the partitioning of the keys being written. This is called *write skew*. Partitions may get hot and cause performance problems for the application.

DB repartitioning means moving the "home" for records. Moving record keys from one partition to another is complex and impacts application availability. Earlier record versions were placed in their old log. After repartitioning new versions must go to a new log. This is difficult in the midst of ongoing updates.

Two-phase commit while repartitioning is complex. Since two-phase commit is designed to be robust after crashes and restarts, the state of the ongoing transaction and its disposition must be durable. When the identity of each participant is bound to a single log, it is clear who to contact to continue the two-phase commit after failures. Ensuring these durable two-phase-commit responsibilities are met usually forces partitioned databases to shut down work as they repartition to clarify recovery responsibilities.

[12]Sometimes, a log is used to ship changes to a backup system, replaying the log and providing read-only access to slightly older record versions. In these systems, updates only happen on the logging server. That is its "home".

5 RETHINKING OLTP DATABASES: A STRAW-MAN ARCHITECTURE

The toy system sketched in this paper trivializes MANY aspects of real systems (e.g., failures, management, etc.).

We propose a new way to think about data & transactions.
It is NOT a complete system design and has many flaws.

MVCC scales better by separating the past from the future! This can be done by organizing record changes first by time and second by key. Consider the past and the future separately:

- **PAST:** Snapshot reads of historic immutable versions are scalable.
- **FUTURE:** Protect against conflicting updates before committing future TXs.

If we independently consider past reads and future conflicts, it is easier to avoid coordinating new writes with older reads.

Most current databases assign a home for each record (e.g., a partition, a block, or a B+tree[8]). This means changes must modify the record at home. It also means that reads must scamper to the record's home to find its value. This is hard to scale.

In contrast, our hypothetical DB organizes changes by commit time first and then by key. We log all changes for a transaction into its worker's log. Atomicity is easy. Locating old versions of each record is a bit harder.

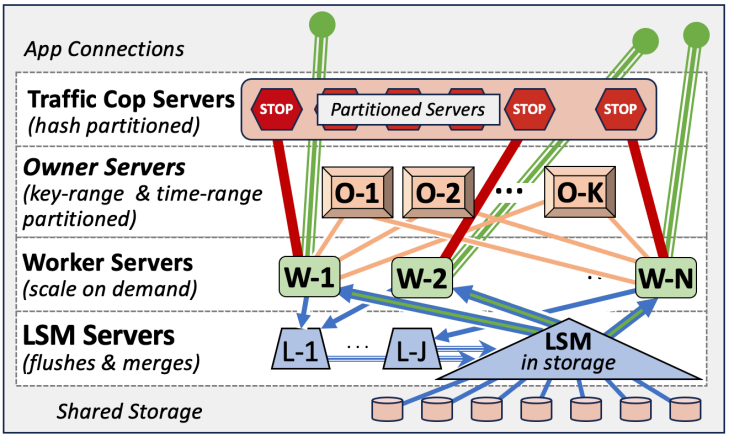

Figure 4: Business summaries roll UP hierarchically over time. *Logarithmic roll up → exponential capacity for changes.*

5.1 A Hypothetical DB: Server Types and Duties

Consider a hypothetical architecture with the following types of servers as shown in Figure 4.

Worker servers each have their own transaction log. They accept connections from app servers, perform transactions & their queries, commit transactions to their per-worker log, and periodically flush committed new record-versions to the LSM. See §5.4.

Owner servers are a scalable set of servers partitioned by both key-range and time-range. Repartitioning happens dynamically to accommodate scale. Each TX is ordered with respect to all other TXs sharing one or more records. Owners verify that concurrent transactions have not created any conflicting updates for each key row-locked or updated by the TX that optimistically hopes to commit. See 5.8. Once committed, changes are visible to any transaction using a later snapshot time. See §5.8.

LSM servers accept flushes from workers and incorporate them into the orderly past stored in the LSM. Record-versions are organized first by time, second by key. By quickly processing committed transactions from workers after they flush, the time between committing a transaction's new record-versions to available in the LSM by key value can be bounded and proportional to the logarithm of the commit rate. See §5.5.

Traffic cops provide pessimistic concurrency control. They will stall later transactions if they acquire a row-lock held by an earlier transaction. This pessimistic ordering of transactions may be violated when failures happen. Semantic correctness will be enforced by optimistic concurrency control prior to commit. See §5.6.

5.2 Centralized TXs Imply Decentralized Data

Centralized transactions offer easy atomicity: When a transaction is centralized in a single log, the atomicity of the transaction is not in question. All changes for each transaction are present in a single log. If you can read the log, you can see the entire transaction and know if it was committed. See §5.4.

Centralized transactions imply decentralized data: To read as-of a snapshot, the latest version of each record before the snapshot must be found. The sequence of record-versions for a record spans multiple logs (from different workers) over time.

Not only does the snapshot's correct version of each key need to be found, the keys present in a key-range as-of the snapshot must be determined. Right after a new record-version's transaction commits, it becomes visible to read with a snapshot time after the commit time. When these changes are new, it's challenging to track their location and make them visible quickly.

5.3 Guessing a Good Time to Commit

Each worker-server and owner-server has its own clock local to the server. These are *approximately synchronized* either by a public cloud provided mechanism or by aligning clock drift[13].

When a worker decides to commit a transaction, it has a pretty good idea of the owner-servers and the partitions touched by the committing transaction. It also has a pretty good idea of the probably latency and clock drift involved in communicating to each of these servers. Based on these guesses, it can select a *future time* to commit the transaction[14].

[13]Clocks can be *probably* kept in alignment via techniques such as described in §4.4 and §4.5 of the Decoupled Transactions paper[19].

[14]This scheme is explained in detail within §4 of [19] as well as in [37].

5.4 Workers: Centralized Atomic Transactions

The database has *a scalable number of worker servers*, each with its own transaction log. As TX load increases, workers are added. Each TX happens at a single worker server. The worker will:

- **Accept incoming connections** from application servers.
- **Execute SQL statements:**
  - *Reads with snapshots* by key or key-range
  - *Row-locks* are acquired using their unique record key,
  - *Update records* by their unique key
- **Commit:** Guess a future time to hopefully commit:
  - *Verify updates don't conflict* with concurrent TXs.
  - *Verify row-locks don't conflict* with concurrent TXs.
  - *Log the transaction's updates & commit record* in the worker's local transaction log.

While the worker does get help from the LSM, owners, and traffic cops, the actual work of the transaction is centralized and changes (for each transaction) are centralized in the worker's log.

5.5 Decentralized LSM for Older Snapshot Reads

LSMs (Log Structured Merge) trees [27] are organized first by time, then by key. They stack recent changes on top of older changes. Reading a snapshot version must read enough of these stacked up versions. Each LSM layer contains record-versions for a band of time. The boundaries of time between the layers are fuzzy and evolve with LSM merges (compactions) [25]. Newer record versions (identified by key) are higher in the LSM and older are lower[15].

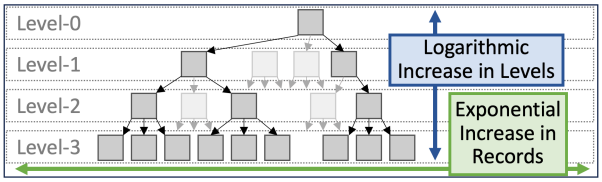

Figure 5: LSMs merge records DOWN the tree in sort order. Lower levels are better for efficient snapshot reads. *Logarithmic roll down → exponential capacity* for stored records.

LSMs are scalable for readers: Because the underlying files (called components) are read-only, it is easy to have many thousands of servers reading them concurrently. Readers gradually evolve their view of the LSM as merges optimize reads but do not stall to coordinate these LSM changes.

LSMs are very scalable for writers: There may be thousands of sources generating new versions of records, each time stamped with transaction commit time. These are coalesced into time windows and layered onto the LSM, organized first by time and second by key. New time-layers are slapped on top once all sources of new committed record-versions are scooped up and sorted by key. These are then made visible to workers so they can read later snapshots.

If N worker servers are contributing new record-versions, a new LSM time-window can be created with N-log-N resources. The time delay (from commit to visible in the LSM) is proportional to Log-N. This is scalable. It gradually coordinates without stalling transactions reading snapshot data. See Figure 5.

With an LSM, the past scales without coordinating across disjoint transactions reading and updating! [25, 27]

[15]The RUM Conjecture [1] describes trade-offs between Read, Update, and Memory to manage data. For asymptotic limits to scale, all of these abound! Only coordination of change is precious. *Maybe we need a CRUM conjecture to include coordination!*

5.6 Decentralized Traffic-Cops

Row-locks have two aspects: *Guaranteeing TX semantics before commit* and *helping coordinate concurrent TXs.*

Traffic cops help coordinate concurrent transactions by stalling later transactions competing for a lock[16]. The *owner-servers* guarantee TX semantics before commit. See §5.7.

Row-locks are requested with an exact key. They ensure at commit time the record's contents aren't concurrently changed and also stall competing transactions to optimize throughput. These stalls help throughput and performance but need not be perfect.

Traffic cops are a decentralized cluster of servers. Since row-locks are by exact-key, the keys can be hashed for lookup. Workers hash the key and send an RPC to the correct server in a consistent hashing[23] cluster of traffic-cops. Traffic-cop servers respond to RPCs from workers to perform traffic cop duty, looking up locks by hashcode[17].

Traffic cop blast radius of failure: Row-locks taken by a TX cause the TX's worker to add an optimistic check prior to committing the TX. *This ensures correct TX locking semantics*[18]. It's OK for the traffic-cop behavior to be inaccurate if it is rare.

5.7 Decentralized Owners: Optimistic Commit

The BIG DEAL's rules are enforced for future transactions before they are allowed to commit at a proposed future time.

Owner-servers enforce the BIG DEALs rules. They verify that updates and row-locks don't conflict with concurrent updates since snapshot time. They also align the TX's commit-time with concurrent changes to the TX's records.

Owner-servers align *commit-time* for records & workers. As a commit-time for a transaction is *guessed by the worker*, every update and row-lock must be verified:

- *Updates:* No conflicting updates from snapshot to commit.
- *Row-locks:* No conflicting updates from snapshot to commit.

If there are conflicts, the transaction may not commit.

Recent changes are verified by the owner-servers. Record-versions not yet present in the LSM are optimistically verified by the owner-servers prior to commit.

Older changes are verified in the LSM by the workers. If a snapshot is old enough, the LSM itself may need to be checked for conflicting updates no longer in owner-servers.

5.8 Decentralized Owners: Update, Lock, & Read

Before commit, the worker sends a *proposed-update* for each update and a *verify-lock* for each row-lock.

Proposed-updates comprise: **[key, proposed-commit-time, snapshot-time, worker-id, & record-value]**
Verify-locks comprise: **[key, proposed-commit-time, snapshot-time, & worker-id]**

Aligning time at the owner-servers: As incoming *proposed-updates* and *verify-locks* arrive, they include a *proposed-commit-time*. Incoming requests from workers hopefully arrive at owner-servers before their local clock has reached the proposed-commit-time:

- *Arrived after commit-time:* Return an error and the TX aborts.
- *Arrived before commit-time:* The owner-server waits until its local clock reaches commit-time.

Aligning the owner-server's local time with the commit-time is essential to the BIG DEAL. Every *proposed-update* and *verify-lock* will be processed after other transactions in time order.

Verifying locks for *verify-lock* requests: As the commit-time arrives at the local server's clock, all records that *may have conflicting updates* will be known. The owner-servers must check back in time to the snapshot-time for the lock being verified[19]. Any updates by concurrent transactions to the exact-key value for this *verify-lock* cause the transaction to abort by declining to verify.

Accepting new proposed-updates: The first step of processing a *proposed-update* is similar to a *verify-lock*. As the commit-time arrives at the local server's clock, all records that *may have conflicting updates* will be known. The owner-servers must check back in time to the snapshot-time for update being proposed[20]. Any updates by concurrent transactions to the exact-key value for this *verify-lock* cause the transaction to abort by declining to verify.

> The existence of a *proposed-update* in an owning partition means the presence or absence of this newly committed record-version is not yet confirmed as committed or aborted.

Resolving proposed-updates: Proposed-updates become either *confirmed-record-versions* or *aborted-record-versions*. Usually, this happens quickly as the transaction's worker tells the owners of the outcome. Sometimes, they may get stuck waiting to hear[21].

Snapshot reads of recent record-versions: Workers may have recent snapshots requiring they see committed updates since that time. Workers can read record versions from owner-servers either by *exact-key* or by *key-range*.

When a snapshot read arrives at an owner-server, it locates any qualifying record-versions by their exact-key or key-range. Any matching records with *unresolved proposed-updates* become a challenge. The snapshot-read cannot be serviced without determining the disposition of the *proposed-update*.

Owner-servers respond to the reading worker describing their dilemma[22]. This can be resolved in a number of ways:

- **Wait for the LSM:** Once the LSM includes records up to *commit-time*, it will know.
- **Ask the worker:** If the worker is alive, it will know.
- **Check the worker's log:** If the worker is unresponsive, the transaction log can be *fenced*[23] ensuring a clear outcome.

[16] This database must support both *scalable applications* and also *not-so-scalable applications.* . Row-locks on disjoint keys should not require cross-transaction coordination.

[17] Hashcode collisions are rare and acceptable.

[18] The servers implementing partitions of the traffic cop for row-locks can be unreliable. Failures of traffic-cop servers can allow reordering of the coordinated TXs. If this happens too often, human users may get annoyed as they lose their partial work. If so, replication of these servers can be added to reduce the likelihood of losing the traffic cop's pessimistic coordination.

[19] We will see below in §5.9 that this is more complex than this initial description.

[20] Complexities processing *proposed-update* requests are also discussed in §5.9.

[21] Unresolved *proposed-update* requests will resolve over time as the LSM advances the committed work it holds. The actual outcome of the *proposed-update* is contained in the worker's log. As time advances, the owner's unresolved state is discarded. Since the *proposed-update* is marked with the *proposed-commit-time*, it has a bounded lifetime.

[22] Both the *commit-time* (if it committed) and the *worker-id* are known.

[23] The updating transaction's outcome is in the worker's log. In our hypothetical DB the log is in shared storage and can be read by another worker. This complex topic is covered in great detail in §15 (Appendix D: Jitter-Free Log Repair) in [19].

5.9 Decentralized Owners: Partitions & Servers

Owners manage this interweaving of updates and snapshot reads for two dimensional rectangles of a *time-range* and *key-range*. Snapshot reads of a key-range may need to union changes back in time spanning owners and the LSM.

Owners are implemented in a scalable set of servers. These servers hold many partitions, each covering a rectangle of key-range and time-range. As new record-versions are added by committing transactions (via *proposed-update* and *resolve-update*), careful management of advancing time and partition capacity is essential.

Owner-partitions can get full and direct new *proposed-updates* elsewhere. Each owner-partition is either:

- **Closed for new business** and accept worker requests for:
  - *Snapshot reads* in their rectangle of key & time ranges,
  - *Proposed updates*[24] &
  - *Notifications of transaction outcome.*
- **Open for new business:** Allowing four kinds of requests:
  - *Snapshot reads* (by key or key-range) after their start-time,
  - *Notifications of transaction outcomes*,
  - New *proposed-updates*, or
  - New *verify-locks*

The set of open-for-business owner-partitions covers all key-ranges for the database and evolves as open partitions get full and close for business, passing off their key-ranges to empty owner-partitions to receive new *proposed-updates* and *verify-locks*[25].

Owners-partitions unilaterally repartition by time & key. At any time, the owner-server for an owner-partition may decide that it is full. It may then redirect new proposed-updates to a different partition that is open for new business. Hence, write traffic can be largely segregated from snapshot-read traffic[26].

Repartitioning decisions can be unilateral with the catalog of owner-partitions being updated later to reflect the new key-ranges and time-ranges for both open and closed partitions. This allows the system to cope with traffic changes.

Blast radius of failure as conflicts are checked leverages the advancement of time and the LSM. If an owner-server holding one or more owner-partitions crashes, it may or may not have its contents replicated. If no replica survives, the system has a gap in its recent history. New attempts to commit a TX might fail. The DB must guarantee no lock or update violations prior to commit.

As the LSM processes committed updates from the set of workers, the maximum time present in the LSM advances. The loss of owner-partitions may mean that transactions accessing that key-range must fail until time advances far enough. This causes temporary loss of availability but only for the affected key-ranges[27].

Blast radius for snapshot reads follows a similar pattern. Loss of owner-partitions may cause a key-range & time-range for snapshots to be unavailable until the LSM time advances[28].

[24] Proposed-updates verify back to their snapshot (possibly visiting closed partitions).

[25] A closed partition can create new replicas of its contents should read traffic scale.

[26] If each owner-partition and the server on which it is placed have pre-allocated empty partitions elsewhere, traffic may be simply redirected as the owner-partition detects a problem. Storage capacity, read traffic, or rate of new *proposed-updates* can cause an open owner-partition to become closed.

[27] Some implementations could choose to add replication to the owner-partitions to reduce the likelihood of this.

[28] Older snapshots for a transaction don't experience outages when owner-servers fail unless they locked records in that key-range. These, too, will recover as time advances.

5.10 Centralized Logging per TX Is Less Brittle

As discussed in §5.2, a DB architecture must decide to either:

- *Centralize TX logging:* Here, all new record-versions for each transaction are logged at the TX's worker.
- *Centralize data logging (i.e., a "home" for each record):* Each new record-version for a key is logged to the same log.

Centralized TX logging simplifies recovery of partitions. As discussed above in §4.3, by centralizing the location of each data record at a well-known "home", data for a TX may be spread across many partitions. This implies the use of Two-Phase-Commit[5, 15, 26] or some similar algorithm. This is especially pernicious when supporting online repartitioning to cope with write skew should the write patterns not match the pre planned partitioning.

Two-Phase-Commit (2PC) is a log-to-log protocol: It may initially seem like a server-to-server protocol but this breaks down as compute and storage are separated. Key-range partitions log in per-partition logs. 2PC must manage the persistent state for both TX participants and TX coordinator. These important 2PC state transitions happen via logged records.

Moving "home" for a record-version makes 2PC complex. If a data record is part of a transaction that is *in-doubt* during 2PC, moving its logged "home" by repartitioning has many subtle races and edge conditions. Which log is part of 2PC and when? In practice, repartitioning is only performed while the partition is offline[29].

6 SCALABLE APPLICATION QUEUING

Can an OLTP system have a queue that scales a million-fold above today's app queues? What must the app and DB do to scale queuing?

6.1 The Ambiguity of Scalable TX Queues

Scalable transactional queues are imperfect. Their order of processing and timeliness are probabilistic.

Any transaction can abort and restart. The processing of the first item in the queue may fail and be requeued for a later time. Since dequeue is approximately ordered, it is not important for enqueue to be perfectly ordered. Assuming perfect order of transactional queues will result in occasional bugs in the application.

Dequeuing at scale can't be in perfect order. Other items in the queue may be dequeued and complete before the restarted item gets another chance. Perfect order doesn't scale past one at a time.

6.2 Choosing Disjoint Keys for Queued Work

Enqueuing work from independent application servers must scale.

A scalable queue has MANY app servers and one DB. Each server enqueues with approximate time order. Enqueued keys should very rarely collide with other keys.

ULIDs (Universally Unique Lexicographically Sortable IDs) are provided by a standard open source library[35]. ULIDs are 26 character keys in approximate chronological order. They combine:

- *48 bits (time):* Local time at the app server [30].
- *80 bits (random):* An 80-bit pseudo random number.

[29] Repartitioning classic cross-partition log-to-log behavior is different than repartitioning owner-partitions in this proposal. Owner-partitions do *NOT* have persistent state. The atomic and persistent state of a TX's outcome is in the single log of the worker and eventually in the LSM.

Owner-partitions can be repartitioned *WITHOUT* needing two-phase-commit. This can be a unilateral decision made by an over-full owner-partition.

[30] ULIDs begin with local time in milliseconds since Jan 1, 1970.

Enqueuing collisions are *extremely rare* and result in a TX abort. As an app server uses a ULID for a queued record's key, the chances of collision are extremely low and result in a transaction abort.

6.3 Scalable Dequeue via SKIP LOCKED

Scalably dequeue at a massive scale is challenging. Let's discuss some techniques combining database and application behavior to empower scalable application queues.

SELECT <query> FOR UPDATE SKIP LOCKED allows the DB to return a subset of the qualifying records (see §2.3). The DB may start looking for qualifying records using an older snapshot[31] with a subset of the SELECTed keys[32].

SKIP LOCKED allows scanning an old DB snapshot. When SKIP LOCKED is used in a SELECT statement, the DB may perform a key-range scan of an old snapshot, even if it includes records that were later deleted[33]. Scalable dequeue can be done by app servers with random exploration of ULIDs:

- Randomly positioning into the SELECTED key-range.
- Find a candidate key in an older snapshot in the LSM.
- Acquire a row-lock on the candidate key.
- Ensure the key has not been updated by any concurrent transactions before commit. Dequeue completes exactly once.

This can be used to return the TOP N keys qualifying for the SELECT and not actively being processed by another application server.

A BIG DEAL DB can scale dequeue transparently. Suppose a dequeuing app server SELECTS with SKIP LOCKED to get queued keys from January 1st, 1970 (the lowest ULID) to the present. The DB is free to omit any or all of the records in the queue[34].

7 MASSIVE SCALE: IT'S ABOUT TIME!

Let's consider how time impacts our scalable database. We discuss the flow of time in a centralized BIG DEAL database, and how it provides snapshots, commits, and external consistency.

7.1 What's External Consistency?

External Consistency[13] ensures new incoming requests see all previously exposed data, even by other database connections. Snapshot reads from new incoming work must be after all committed work *previously visible outside the database.*

Ensuring external consistency gets more complex as the geographic scope of a DB grows past a single server or datacenter. Pressure rises on the timeliness of snapshots seen for distant data as well as optimistic conflict checking at commit time.

[31] This hypothetical DB moves older committed record-versions into its LSM in shared storage. The read only LSM files may be replicated as needed ensuring scalable scans over key-ranges such as ULIDs.

[32] Verifying the older records have not been modified since the record-version in the LSM uses the exact-key of the record to access the owner-partitions. Similarly, checking the traffic-cop servers is by exact-key. Both scale very well.

[33] SKIP LOCKED allows the DB to omit any records qualifying for the SELECT if it so chooses. Ignoring newer records not yet in the LSM is OK.

[34] The DB could approximately track the key distribution of the ULIDs and randomly position into dense key ranges with a slight bias for the older entries.

To minimize the impact of concurrent processing of a key can be provided by checking both the traffic-cops and owner-partitions. In this way, a torrent of application servers can pummel the queue and most will get work to process without colliding with the rest of the gaggle. If dequeuing a key deletes it, each request is transactionally processed exactly-once.

7.2 The Order of Time in Our Hypothetical DB

The interweaving of *partial order* and *total order* empowers scale for our hypothetical DB. First, let's consider when time must be aligned using *partial order* for work distributed across servers:

- **Snapshots assigned to new incoming work** are AFTER all earlier committed TXs *visible outside the DB.*
- **Commit time for a TX** is AFTER the TX's snapshots.
- **Committed record-versions for a key** are AFTER its earlier record-versions and BEFORE its later record-versions.
- **Visibility to the app** exposes the committed TX outside the DB. It is AFTER the commit-time and may be delayed[35].

These time alignments are easy in a centralized DB and more challenging in a distributed DB!

Time is totally ordered across TXs. Commit-time comprises:

- The time the worker *guessed* for a future commit, and
- A suffix with the worker-id.

These *unique commit-times* have total-order. Each separate key within the TX has the same commit-time.

Time is totally ordered across record-versions for each key. Each record (by key) has a sequence of record-versions labeled with their *commit-time*. Time for the key advances one record-version at a time. The most recent record-version may be unresolved (neither committed nor aborted) until the outcome of its TX is determined. Time marches forward for each key, one record-version at a time.

Time is totally ordered across a key-range's record-versions. Each key-range comprises a set of keys. As new record versions for keys within the range see *proposed-updates*, the time-range for the key-range expands[36]. Time-ranges have a *minimum lower-time* and a *maximum upper-time* that includes unresolved *proposed-updates*.

Time is partially ordered across owner-partitions. Owner-partitions may be *open-for-business* or *closed-for-business*. For any TX sending a *proposed-update* as a part of optimistic commit, there is exactly one *open-for-business owner-partition* to use[37]. For any TX sending a *snapshot read*, a partially ordered time-ordered sequence of owner-partitions must be accessed. These move from the snapshot time of the read backwards across one or more time-ranges. Eventually, the time-ranges to search backward in time transition to the LSM and its levels[38].

Time in our hypothetical DB is relative:

- **Total-order** of committed TX.
- **Total-order** of snapshots seen within each TX.
- **Total-order** of record-versions per key.
- **Partial-order** across owner-partitions.
- **Partial-order** for external consistency:
  - Commit visibility AFTER commit-time.
  - Snapshots for incoming work AFTER visible commits.

[35] Delaying visibility expands the time between *commit* and *visibility outside the DB.*

[36] Unresolved *proposed-updates* have a well known *proposed-commit-time*. Until they are resolved, the time-range includes these possible new record-versions, expanding the time-range as needed.

[37] Actually, there is a window of time when the old open-for-business owner-partition is closing and handing off responsibility. These may both be open-for-business and have overlapping time-ranges. The older partition responds to check both for conflicts.

[38] Reading a key-range by snapshot-time means including the latest record-version for each key in the key-range that is at or before the snapshot-time.

7.3 Time & Order: Centralized to Distributed DB

As a BIG DEAL database transitions from a centralized DB to our massively distributed hypothetical DB, these time and order guarantees must be preserved. Consider two transactions, T1 and T2. T1 commits and is seen outside the DB before T2 arrives as new work. All snapshots of T2 must see the updates made by T1:

- *Visibility(T1) is AFTER commit(T1)*
- *IF New-snapshot(T2) is AFTER visibility(T1)*
  THEN New-snapshot(T2) is AFTER commit(T1)

In a centralized database, snapshot times and commit times are tightly controlled by a single server's clock. Time emanates from a single point in space:

- *Commit time* is grabbed from the server's clock.
- *Visible commit time* is:
  AFTER sending out committed work
  AFTER its commit-time.
- *Snapshot time for new work* comes from the server's clock.

For distributed DBs, snapshots for incoming work must still be AFTER all visible committed TXs, even from distant servers.

Stretching Space Requires Stretching Time
As the database includes more servers, more time is needed between Commit(T1) and Visibility(T1)
Delaying the DB's post-Commit(T1) response over its DB connection stalls Visibility(T1).
The time delay between Commit(T1) must be coordinated with advancement of snapshots throughout the database.

The source of change moves from a point to a cylinder. As the set of servers grows from one to many, it's like moving from a point to a circle surrounding the set of workers committing changes. To align their snapshots and external visibility, time also stretches.

7.4 The Turbulence of NOW

Both databases and applications experience challenges immediately after changes happen:

- **Applications need time** to aggregate their business values.
- **Databases need time** to separate snapshot reads & updates.

For our hypothetical database, the *turbulence of now* happens at *open-owner-partitions*. These partitions support many keys in the same key-range and must accept:

- ***Snapshot-reads***
- ***Verify-locks***, and
- ***Proposed-updates***

This introduces the risk of *coordination across disjoint transactions*, the official nemesis of our hypothetical scalable database.

Unilaterally repartitioning to survive *the turbulence of NOW*: Owner-servers with *open-owner-partitions* should preallocate empty partitions on other owner-servers. These are available to accept new traffic. When *proposed-updates* cause too much traffic, the *open-owner-partition* may "close for business". It is then almost read-only[39]. *Closed-owner-partitions* may be replicated for scale.

[39] *Transaction-Outcome* requests do modify closed partitions. Should a closed partition be replicated, it's OK for a *Transaction-Outcome* request to update only one replica. If needed, the outcome can be rediscovered. *Snapshot-reads* and *Verify-lock* requests are also supported. None of these requests to closed partitions require exclusively locking in-memory data structures. Hence, they don't coordinate across disjoint TXs.
The nemesis is defeated!

8 RELATED WORK

Many papers discuss scalable systems. Most don't discuss limits to scale for today's typical SQL DBs and their apps. This paper examines the semantics of the BIG DEAL (or SQL + RCSI) as the domain specific OLTP application works cooperatively with the database. It points out the incredible power of snapshot isolation with MVCC (Multi-Version Concurrency Control).

Ziegler et al[38] prompted this work by asking:

"What are the asymptotic limits to scale for OLTP systems?"

Their paper describes three archetypes for existing DBs: single-writer, partition-writer, and multi-writer. Each of these existing approaches assumes every record has a "home" for its current version. They did not consider providing MVCC by separating reading older versions from checking the validity of newer changes.

New scalable models of storage and DBs have been proposed in other work. Many of these are forms of key-value stores[6, 7, 10, 24, 31] that do achieve large scale, albeit not with a SQL interface.

Partitioned writer SQL databases[2, 9, 14, 33] have been around in various forms for many years. Each of these suffers from *write-skew* as the transaction traffic updating the database may not match the static or semi-static partitioning.

Workflow style business functionality for scale has been proposed[18]. This approach is not transparent support for OLTP.

Replication approaches for both offline and for scale have been considered in numerous papers. Shapiro et al[30] describes CRDTs (Commutative Replicated Data Types). Hellerstein et al[20] describes CALM (Consistency as Logical Monotonicity) and shows how monotonicity provides a framework for to provide deterministic behavior from concurrent non deterministic. Bailis et al[3] describe how to map SQL operations over replicated data to merge their state into a form of a materialized view.

Each of these is addressing a different problem by looking at the system as independent replicas, not a unfied OLTP system.

This work considers the impact of today's BIG DEAL. What are the limits to scale that can be achieved compatibly with today's model for OLTP systems? To our knowledge, this is the first paper to point out the inherent scaling challenges with today's implementations of MVCC. We also try to conceptually describe what is needed to build a scalable application.

9 ACKNOWLEDGEMENTS

I'd like to thank my colleagues at Salesforce: Jamie Martin, Thomas Fanghaenl, Jim Mace, Ben Busjaeger, Bryan Pendleton, Simon Wong, Xinan Yan, Shyam Antony, Atish Agrawal, Dave DeHaan, and Nat Wyatt for their comments and discussions through the years. The anonymous CIDR reviewers were very helpful. I am grateful for the discussions with my friends Anastasia Ailamaki, Murat Demirbas, Aleksey Charapko, Chuck Carman, and Shel Finkelstein. The support of Subho Chatterjee and Roy Chowdhuri has been invaluable.

Thanks to Michael Abebe for his review and insights. Special thanks to my friend and colleague Daniel May for many hours of help on this paper.

10 CONCLUSION

We've discussed possible future directions for DB and app research. What got us to this point and where can it take us moving forward?

Our *gedanken experiment* asks about asymptotic limits to scale for OLTP systems focusing on the BIG DEAL[40]. This has led to some new insights into scale for both applications and databases.

The BIG DEAL narrows coordination. The gradual evolution of SQL database isolation semantics has provided major steps forward (on scale) and backward (on isolation). It has done so by narrowing the places at which transactions must coordinate.

The BIG DEAL clarifies expectations for the DB & the App. A scalable DB must guarantee certain behavior to the application including promising scalable execution of scalable operations (e.g., snapshot reads). Existing RCSI applications must already tolerate some subtle behavior from the database. If they don't they have problems today. Our thought experiment leverages these existing subtleties empower more scale in the database implementation.

The turbulence of NOW! Recent changes are more difficult to understand than older changes. This is true for the application as it changes business state. It is also true for the database as it layers recently committed record-versions into visible snapshots.

Layering changes by time helps solutions scale. Narrower time slices can bound the turbulence of NOW. In applications, aggregating business state over a sequence of narrow time windows reduces *the turbulence of NOW!* [41]. In databases, quickly segregating new changes from read-only historic state reduces *the turbulence of NOW*[42]. This can be accomplished by layering potential changes in time-order on top of their predecessors.

SELECTs requiring key-ranges are one of the more challenging aspects to RCSI databases. Older snapshots are easier in LSM style systems as they span fewer levels. The older, the better[43].

10.1 Evolving Apps & DBs for Scale

The proposed DB design has many issues, potentially fatal ones. However, unlike partitioned DBs, it does not suffer from write skew[13] as it seamlessly adapts to workload changes. We should revisit our community's tacit assumption that the "current" value of a record has "a place to call home".

Similarly, as applications scale they should rethink concentrating the aggregated values of business state in dedicated records. By asynchronously aggregating, they can offer more scale.

10.2 Taking Time as You Slowly Share

I grew up with 3 brothers. None of us were in a hurry to share. Scalable systems must not be in a hurry as they share resources.

Coordination is painful. It hurts to share across records in the application, data structures in the database, or servers in a scalable system. Sharing requires patience. You certainly don't know when your brother will return what he's borrowed!

Exponential scaling requires logarithmic consolidation. Increasing rates of change can become overwhelming, either in an application or database. Can these cope with a million-fold increase? A hierarchy to consolidate changes avoids overwhelming any part of the system. The depth of the hierarchy is proportional to the logarithm of the scale as work is consolidated in stages.

Logarithmic consolidation can flow UP and DOWN. Application consolidation flows UP a hierarchy as business values are aggregated over time. Database consolidation flows DOWN a hierarchy as an exponentially growing number of changes are organized to be read by an exponentially growing set of keys. LSMs are one way to provide logarithmic management of exponential growth. See Figures 3 and 5.

10.3 Conclusions Reached in This Paper

Our introduction (§1.3, §1.4, §1.5, & §1.6) presented this paper's conclusions in some detail. We summarize these conclusions here.

#1) The BIG DEAL (SQL + RCSI) semantics are scalable.
The BIG DEAL splits scaling responsibilities:

- **Scalable apps** don't concurrently update the same key.
- **Scalable DBs** don't coordinate across disjoint TXs.

#2) The BIG DEAL has guarantees & weaknesses.
The SQL RCSI interface has two aspects seen by databases:

- **Guarantees** that the DB must provide to the app.
- **Weaknesses** that are tolerated today by RCSI applications ensuring the DB may exploit them.

#3) Domain specific apps change behavior as they scale.
An application's domain specific behavior weakens as it scales:

- **Future commitments are fuzzier:** Human users get weaker promises.
- **Past biz state is blurry at first & gets clearer over time:** The nature of this blurriness depends on the application's business domain.

#4) Scalable DBs have no "current" data, only snapshots.
Organizing data by its creation time empowers scalability:

- **Updates create new record-versions** and should not coordinate with readers.
- **Reads are always by snapshot time** and should not coordinate with updates.

Today's Databases Don't Scale!

Reads & writes fight to access the "current" record-version on the way to the snapshot's record-version.

[40] The existing SQL + RCSI with SKIP LOCKED semantics.

[41] Of course other partitioning of business state (e.g., by region, management chain, or product type) helps, too.

[42] Of course, additional partitioning by key helps, too.

[43] As someone with 45+ years working on database implementations, "older is better" can be a comforting thought.

REFERENCES

[1] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. https://doi.org/10.5441/002/edbt.2016.42

[2] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickenson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. *SIGMOD 2017 Proceedings of the 2017 ACM International Conference on Management of Data* (May 14th-19th 2017), 331–343.

[3] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems (Extended version). *arXiv preprint arXiv:1402.2237* (2014).

[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.* 24, 2 (may 1995), 1–10. https://doi.org/10.1145/568271.223785

[5] Philip A. Bernstein, Philip A. Bernstein, and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221. https://doi.org/10.1145/356842.356846

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. https://doi.org/10.1145/1365815.1365816

[7] Shanshan Chen, Xiaoxin Tang, Hongwei Wang, Han Zhao, and Minyi Guo. 2016. Towards scalable and reliable in-memory storage system: A case study with Redis. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 1660–1667.

[8] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (jun 1979), 121–137. https://doi.org/10.1145/356770.356776

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heider, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura978-0-321-84268-8, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. *Tenth USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)* (October 8th - 10th 2012), 251–264.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[11] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (nov 1976), 624–633. https://doi.org/10.1145/360363.360369

[12] Dieter Gawlick and David Kinkade. 1985. *Varieties of Concurrency Control in IMS/VS Fast Path.* Technical Report. https://www.hpl.hp.com/techreports/tandem/TR-85.6.pdf

[13] David Kenneth Gifford. 1981. *Information storage in a decentralized computer system.* Stanford University.

[14] J Gray et al. 1987. NON-STOP SQL. In *Proc. 2nd International Workshop on High Performance Transaction Systems, Asilomar, CA.*

[15] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann. 1070 pages.

[16] J. N. Gray, R. A. Lorie, and G. R. Putzolu. 1975. Granularity of Locks in a Shared Data Base. In *Proceedings of the 1st International Conference on Very Large Data Bases* (Framingham, Massachusetts) *(VLDB '75)*. Association for Computing Machinery, New York, NY, USA, 428–451. https://doi.org/10.1145/1282480.1282513

[17] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. 1994. *Granularity of Locks and Degrees of Consistency in a Shared Data Base.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 181–208.

[18] Pat Helland. 2017. Life beyond distributed transactions. *Commun. ACM* 60, 2 (2017), 46–54. https://doi.org/10.1145/3009826

[19] Pat Helland. 2022. Decoupled Transactions: Low Tail Latency TXs Atop Jittery Servers. CIDR 2022. https://www.cidrdb.org/cidr2022/papers/p5-helland.pdf

[20] Joseph M Hellerstein and Peter Alvaro. 2020. Keeping CALM: when distributed consistency is easy. *Commun. ACM* 63, 9 (2020), 72–81.

[21] IBM. [n. d.]. IBM Information Management System. https://www.ibm.com/products/ims

[22] IBM-IMS:Wiki [n. d.]. Wikipedia: IBM IMS (Information Management System. https://en.wikipedia.org/wiki/IBM_Information_Management_System.

[23] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Symposium on the Theory of Computing*. https://api.semanticscholar.org/CorpusID:263889166

[24] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[25] Chen Luo and Michael J. Carey. 2019. LSM-based storage techniques: a survey. *The VLDB Journal* (Jul 2019). https://doi.org/10.1007/s00778-019-00555-y

[26] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (dec 1986), 378–396. https://doi.org/10.1145/7239.7266

[27] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[28] Oracle. 2011. Oracle DB Concepts 11g. https://docs.oracle.com/cd/E25054_01/server.1111/e25789.pdf.

[29] PostgreSQL 2023. PostgreSQL 15.3 Docs. https://www.postgresql.org//files/documentation/pdf/15/postgresql-15-US.pdf.

[30] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.

[31] Swaminathan Sivasubramanian. 2012. Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 729–730. https://doi.org/10.1145/2213836.2213945

[32] SQL Server 2023. SQL Server Isolation. https://learn.microsoft.com/en-us/sql/t-sql//t-sql/language-reference?view=sql-server-ver16.

[33] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[34] TPC: Transaction Processing Performance Council [n. d.]. https://www.tpc.org/information/about/history5.asp

[35] ULID [n. d.]. ULID: Universally Unique Lexicographically Sortable Identifier. https://github.com/ulid/spec.

[36] Wikipedia contributors. 2023. Einstein's thought experiments — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Einstein%27s_thought_experiments&oldid=1143806192 [Online; accessed 30-July-2023].

[37] Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: Using Network Measurements to Reduce State Machine Replication Latency in WANs. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/3386367.3431291

[38] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem?. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p50-ziegler.pdf