WiSer: A Highly Available HTAP DBMS for IoT Applications

Ronald Barber¹, Christian Garcia-Arellano², Ronen Grosman², Guy Lohman¹, C. Mohan¹, Rene Muller¹, Hamid Pirahesh¹, Vijayshankar Raman¹, Richard Sidle¹, Adam Storm², Yuanyuan Tian³, Pinar Tozun¹, Yingjun Wu¹

¹IBM Almaden Research Center  ²IBM Analytics  ³IT University of Copenhagen  ⁴Bern University of Applied Sciences BFH

ABSTRACT

In a classic transactional distributed database management system (DBMS), write transactions invariably synchronize with a coordinator before final commitment. While enforcing serializability, this model has long been criticized for not satisfying the applications’ availability requirements. When entering the era of Internet of Things (IoT), this problem has become more severe, as an increasing number of applications call for the capability of hybrid transactional and analytical processing (HTAP), where aggregation constraints need to be enforced as part of transactions. Current systems work around this by creating escrows, allowing occasional overshoots of constraints, which are handled via compensating application logic.

The WiSer DBMS targets consistency with availability, by splitting the database commit into two steps. First, a PROMISE step that corresponds to what humans are used to as commitment, and runs without talking to a coordinator. Second, a SERIALIZE step, that fixes transactions’ positions in the serializable order, via a consensus procedure. We achieve this split via a novel data representation that embeds read-sets into transaction deltas, and serialization sequence numbers into table rows. WiSer does no sharding (all nodes can run transactions that modify the entire database), and yet enforces aggregation constraints. Both read-write conflicts and aggregation constraint violations are resolved lazily in the serialized data. WiSer also covers node joins and departures as database tables, thus simplifying correctness and failure handling. We present the design of WiSer as well as experiments suggesting this approach has promise.

1. INTRODUCTION

For decades, database management systems (DBMSs) have separated on-line transaction processing (OLTP) from on-line analytics processing (OLAP) in distinct systems. This has been more of an engineering compromise than a desirable feature: OLTP systems typically modified small portions of the database and demanded low latency, whereas OLAP systems typically queried large swaths of the database in read-only mode and demanded high throughput. But the promise of classical DBMS transaction theory and its serializability concepts is that all transactions – even with arbitrarily complex queries embedded within them – can be handled correctly (i.e., serializably) by a single DBMS, so that applications can program against a single-system image, even while running in a distributed system with high concurrency.

Increasingly over the last few years, the Internet-of-Things (IoT) applications are gaining their popularity. They have begun to demand this ideal of doing analytics on the same system as transaction processing, and even doing analytics within the transactions themselves. This is known as hybrid transaction and analytical processing (HTAP). However, even with significant advances in the speed and capacity of the underlying hardware, true HTAP in a distributed environment is extremely challenging. If a transaction, running on any node, can query the entire database state up to that point in time (in serializable order) and subsequently make a modification that might touch the entire database, then that transaction will effectively need to lock the entire database while it runs. Not only can such transactions run for a long time, but their locking behavior excludes other transactions, killing concurrency and slowing performance to a crawl.

Database sharding across distributed nodes somewhat eases the problem by partitioning the database into shards and similarly partitioning the transaction workload, so that most transactions are local to a single shard. But often aggregation constraints spanning multiple nodes need to be enforced, and this usually forces escrowing transactions while aggregating across shards.

Let us consider an online shopping application that is at the core of the TPC-C and TPC-E transactional benchmarks and captures the full HTAP challenge. It has two tables:

- Products(productId, unitPrice): containing real goods (as in TPC-C), securities (as in TPC-E), or services (e.g., a slot on a delivery truck).
- Orders(orderId, productId, qty, price): containing orders for a given quantity of a given product, at a given price. Customer orders express quantity as a negative number, while restocking orders have positive values for quantity.

Now consider we want to run the following two transactions on these tables:

- NewOrder: Look up current prices for a few products, and insert rows into Orders table with a certain order ID.
- UpdatePrice: Change the unit price for some products in the Products table.

Note that, unlike TPC-C, we do not have an “Inventory” table holding a materialized running overall quantity of product; instead, the transactions need to check an aggregation constraint ensuring that the sum of all quantities cannot be negative for any product. The lack of “Inventory” table is common in modern IoT applications. As NewOrder can come from edge devices (e.g., mobile phones) from across the world, maintaining such an “Inventory” table could easily lead to centralized contention, caused by both 2-phase locking (2PL)
and 2-phase commit (2PC). In such an application, it is impossible to escrow the aggregation constraints; in addition, it is also not realistic to shard the database, as these aggregation constraints are not “shardable”.

Instead of maintaining aggregation views in the DBMS, these constraints are invariably delegated to application logic. This brings us to the limitations expressed by the well-known CAP theorem by Brewer, which states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees: Consistency (C), Availability (A), and tolerance to Partitions of the network connecting nodes (P).

1.1 CAP Theorem within DBMS

Brewer observes that applications always prioritize availability, even applications that modify complex distributed state with many concurrent actors \[8\]. A transaction cannot always wait to access the full global state and cannot synchronously modify this state in a way that is immediately reflected globally. Instead, there is complex compensatory logic in the application to handle constraint violations due to concurrent activity.

A transaction is a contract between two parties, e.g., between a customer and a retailer. When the customer clicks the “submit” button, they are committing, not just in the DBMS sense but also in the legal contract sense, to buy these products at these prices; conversely, the retailer is committing to supply.

But, the serializable order of these transactions is established after commit. In a retail application, this could be the sequence in which orders are filled by consuming inventory or space on a delivery truck. Constraints can and do get violated all the time, and are resolved by compensation, e.g., via an apologetic email about a “back-ordered” item and perhaps a coupon.

Thus, commit always has “asterisks” (contingencies): the parties agree that the contract is contingent on the global state (reflecting all concurrent transactions, in serializable order) not differing too much from the state seen at commit. Banks usually state this asterisk explicitly, e.g., by listing their algorithm for processing overdrafts. The problem with this application logic is that it complicates reasoning about database atomicity or durability.

1.2 HA and HTAP within DBMS

The Wildfire-Serializable (WiSer) project seeks to provide both high availability (HA) and HTAP within the same distributed DBMS without sharding, thus extending atomicity and durability to the handling of consistency contingencies. WiSer is a follow-on project to the Wildfire research project, which developed high-throughput HTAP at snapshot isolation and has been commercialized as the IBM DB2 Event Store \[5\]. Now we first give an overview of the novel way in which WiSer commits a transaction to achieve this significant improvement.

1.2.1 Symbolic and Resolved Commit

A transaction in WiSer can run on any node, even on poorly connected nodes. Commit is split into two stages:

- **Promise** stage, after which the deltas (i.e., state changes) of a transaction X are persisted durably and the transaction is “symbolically committed”. There are three “symbolic” entities in the delta that are resolved lazily: (1) the position of X’s delta in the serializable order; (2) X’s conflict status – whether the queries in the transaction saw the latest state (per the serializable order); and (3) X’s constraint status – whether the database state up to X satisfies aggregation constraints. These symbols capture the contingency semantics: if X confronts access conflicts or violates constraints, X will roll back and the application can specify one or more compensatory actions.

- **Serialize** stage, to pick the position of the X’s delta in the serializable order, which is done via consensus. This stage inherits the HA and liveness properties of that consensus protocol.

WiSer has further asynchronous stages, to resolve X’s conflict violation, to resolve X’s constraint violation, and to “publish” X’s delta in a query-efficient form. But X’s status is deterministic after the SERIALIZE stage. Thus, an application running on a poorly connected node, as long as it can get to serialize, can always run a query to find the transaction status.

1.2.2 HA HTAP via Consensus

Conceptually, doing HTAP with HA is just a consensus problem. Imagine a database that is just a huge Raft log. Transactions append deltas to this log. In addition to these changes to user tables, each delta must also capture additional metadata and cluster state in this log:

- Transaction snapshot and read-sets: the queries in a transaction X are run as of a snapshot, which is a prefix of this log. To resolve X’s conflicts, we need to know this snapshot and the nature of X’s queries to compute whether the query result would have changed due to deltas between the snapshot that X sees and the position of X’s delta. Often this “read-set” information is stored in custom data structures and lock tables, making it harder to reason about correctness in failure cases.

- Node departures and (re-)joins: say a node N hangs for a while, and so its peers evict it. How do they do that, while atomically stopping that node from adding deltas to the log? We need to add the “node departure” event to the log as well.

![Diagram](image-url)

**Figure 1:** If all state changes (database and system) are in a durable, consistent, HA consensus log, we have a serializable system.

**Figure 1** illustrates this data model. If we had such a consensus log, then in theory we have HTAP and HA – transactions will run serializable just by querying the log, and they will make progress even in poorly connected situations, following the HA behavior of the consensus protocol. Of course, such a grand consensus log would be too slow, both in throughput and latency. This in turn would cause too many rollbacks, because longer-running transactions are more likely to have read-write conflicts.

WiSer instead picks the other extreme. As Figure 2 shows, WiSer has a consensus log that tracks only the history of the state changes. All the other state is in database tables, and our challenge is to get consensus on changes to this other state using only the change log.

1.2.3 High Throughput Scale-out

WiSer’s design of resolving conflicts and constraints after serialize also gives a significant throughput benefit.

**Promise** is a local operation, with replication for durability alone, and does no locking, so it scales out quite readily; **Serialize** is a
consensus operation, whose cost is independent of the transaction load. We have an elected SERIALIZER (or leader), who appends the log-sequence number (LSN) ranges to a broadcasted file every \( N \) time intervals, where \( N \) is chosen based on the desired latency. Each serialization picks the serial order for all the new transactions, across all nodes, that are available to SERIALIZE at that time.

Thus, the key scalability challenge is in the conflict resolution step performed after SERIALIZE, to resolve the symbolic fields. Like Calvin [29], WiSer’s database state is deterministic upon SERIALIZE. So constraints and conflicts are resolved on a non-changing database state. This allows conflict resolution to be done at high throughput, and to be trivially scalable.

### 1.2.4 Efficiency of Aggregation Constraints

In the previous online shopping application, suppose NewOrder transaction had to compute the inventory (via an aggregation query) and verify that it will remain positive after this order. The query will take so long to run that the resulting transaction will surely stall. Thus, we call the deltas produced at each node as a subchain and the database chain is an interleaving of these subchains. WiSer performs background replication and hardening of each subchain, as described later.

#### Figure 2: WiSer puts only the history of state changes in a Raft log, and stores all other information in database tables.

![Raft Diagram](image)

#### Figure 3: Transactions run at three nodes. A transaction only see deltas made before their snapshot.

![Transaction Diagram](image)

**DATABASE AND SYSTEM STATES**

<table>
<thead>
<tr>
<th>Time</th>
<th>Deltas @ Node A</th>
<th>Deltas @ Node B</th>
<th>Deltas @ Node C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \mathbf{A}_1 )</td>
<td>( \mathbf{A}_3 )</td>
<td>( \mathbf{A}_2 )</td>
</tr>
<tr>
<td></td>
<td>( \mathbf{B}_1 )</td>
<td>( \mathbf{B}_2 )</td>
<td>( \mathbf{B}_3 )</td>
</tr>
<tr>
<td></td>
<td>( \mathbf{C}_1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Content of a Delta.** WiSer allows only one kind of state modification: an upsert of a row to a table. An update-by-key is treated as a query to get the prior row content, followed by an upsert of the modified contents. Deletes are updates that set an isDeleted field. A transaction \( X \) can run many queries, and do many (or zero) upserts. The delta encapsulates just these upserts, but also the read-set of \( X \) – the read queries the transaction \( X \) ran. Logically, the queries in \( X \) should read the database chain up to \( X \); that is, \( X \) shall see any updates that happen before it. But in practice, the queries in \( X \) read a prefix of this chain, called \( X \)'s snapshot: as shown in Figure 3, \( B_3 \) can observe the delta made by \( A_1 \), but does not see that made by \( A_2 \), which happens after the snapshot. If these query results can change due to intervening deltas (between \( X \)’s snapshot and \( X \)), \( X \) must roll back. Therefore we add the read-set and the snapshot, as a serialization sequence number, to the delta.

WiSer represents transaction deltas in log format, as a Parquet [1] row-group per modified table, all concatenated together into a byte-string. Read-sets are represented as upsets against a ReadWriteSet table, whose format we explain in Section 3.

#### Figure 3: Transactions run at three nodes. A transaction only see deltas made before their snapshot.

**Consensus for Serial Order of Writes.** WiSer uses Raft [23] for achieving consensus on this serial order. But we do not directly put these deltas, or even the metadata, such as the range of log sequence numbers (LSNs) they span, onto a Raft log. Instead, our Raft log only contains one entry each time a new SERIALIZER (or leader) is elected. This log holds a Serializers table, with schema (SerializersNodeID, SerializersSeqNum, StartingBatch), which is initialized to a single row \((0,0,0)\) at database creation (e.g., node \( A \) is the first leader). Nodes are assumed to have unique IDs. SerializersSeqNum is an increasing number \(0,1,2,..\) which is increased when a new SERIALIZER is elected, and StartingBatch is an offset into a separate SerializersFrontiers table.

SerializersFrontiers holds the serial order of deltas. Each row corresponds to a serialization batch, and contains the node ID originating a batch, and the LSN range (only the upper bound) of the deltas in that batch. Nodes repeatedly run batches of transactions, harden their deltas, and send the LSN range to the current SERIALIZER. The SERIALIZER picks the serial order and records it.
by appending to SerializeFrontiers table. By inspecting the tail of SerializeFrontiers table, a query can determine what is serialized across the whole database.

SerializeFrontiers table is implemented as a list of files: a Serializer with node ID N and sequence number S writes to file named SerFront.N.S (this file is lazily replicated to all nodes). Upon a failure and/or a network partition, multiple nodes can think they are the Serializer, but the Raft log Serializers picks the winner – the row with maximum SerializerSeqNum, with ties broken by picking the earliest entry.

Visibility and Conflict Rollbacks. Of course, transactions do not just do blind writes. They run queries (including both reads and writes) as well. As explained in Figure 3 all queries within a transaction see a common prior state, called its snapshot. This raises two challenges:

- Visibility: the transaction snapshot needs to be recent enough (not too stale), highly available (when individual nodes are down), and often needs auxiliary structures (indexes) to perform acceptably.
- Read-write conflicts: after determining the serializable order (via consensus), WiSer need to roll back all transactions whose reads have been changed in value between the seen snapshot and the serialization point. As discussed above, WiSer append the read-set of each transaction to its delta. This allows precise determination of read-write conflicts.

Section 3 describes how we address both these challenges. Note that we do not call out write-write conflicts (concurrent transactions updating rows with the same key) separately. In WiSer, a write of a row into a table with a defined primary key automatically performs a read of that row (i.e., all writes are upserts), thus handling all conflicts as read-write.

Constraint Rollbacks. State modifications must also respect aggregation constraints, such as the rule in the online shopping application that inventory for each product must stay positive. We require aggregation constraints to be algebraic (that is, involving sums, counts, and averages, but not medians), so as to allow parallel as well as incremental computation.

Even with algebraic aggregates, constraint resolution is challenging because aggregates are generally not monotonic. For example, a batch of transactions may start with some orders that exceed the available inventory, and then a restocking, and then some more orders. Section 4 describes how we tackle this challenge, and presents a parallel algorithm for incremental constraint resolution.

2.1 Lazy Transaction Resolution

The design presented above involves a bunch of background tasks that happen in parallel with transaction execution: replicating deltas, serializing them, resolving conflicts, and resolving constraints. As these background tasks progress, they take a transaction through various stages, which we now detail.

Stage 1: Promise. A transaction submitted by a client is first assigned to a subchain, and accordingly sent to a node. Clients can give a hint as to where to run a transaction, that is, based on what rows the transaction is inserting, to get a partitioning that queries can exploit. But this is just a hint: every transaction can modify the entire database.

The transaction delta is appended to a node-local log, which is implemented as a lock-free list of blocks. WiSer asynchromously and automatically replicates this log to replica nodes. Promise completes when a quorum of replicas has received and hardened the log, which is tracked via heartbeats. Promise needs no global cluster communication; AP applications stop at this stage.

Stage 2: Serialize. Nodes continually inform an elected Serializer of their log growth in the form of LSN ranges. The Serializer repeatedly picks a node and publishes its recent delta to the global serial order, as discussed earlier. Other serial orders are also acceptable, such as ordering by the local commit timestamps. This policy serializes entire batches of changes from each node in a deterministic manner, similar to Calvin [29].

The database state up to the serialization frontier is deterministic. Each delta has two symbolic fields:

- Conflict status: Transactions may roll back, due to read-write conflicts. But since the read-set is embedded in the delta, this resolution is deterministic. We use a table called Rollbacks (appended to by the ConflictResolver), to cache this resolution.
- Constraint status: The ConstraintResolver lazily checks aggregation constraints and appends to a ConstraintFailures table. Queries use these two tables, which are essentially materialized views, to efficiently view the latest serialized database state.

Applications requiring strict serializability can stop at SERIALIZE stage. WiSer does wait for publish and constraint resolution so that it can give a transaction status to the client, but this can time-out. The semantics upon such a time-out is identical to that of time-out at commit in a classical DBMS.

Stage 3: Conflict Resolve. A background ConflictResolver uses the read-sets embedded in transaction deltas (across all nodes) to check if the value of any of these reads has changed by the time the transaction was serialized. In our current implementation, the Serializer performs the role of ConflictResolver as well.

Stage 4: Published. A background Publisher runs at each node to make the serialized data efficiently accessible to point queries.

Stage 5: Constraint Resolve. A background ConstraintResolver continually reads the conflict-resolved deltas, across all nodes, to verify aggregation constraints. It is at this point that the transaction status can be returned to a client.

Stage 6: Groom to Cloud. Publisher writes out very small blocks, because we need low latency. This is too expensive for OLAP, so we perform another round of data reorganization to consolidate blocks and to migrate them to (less expensive) cloud storage.

2.1.1 Strict Serializability

Applications may expect more than serializability: if a transaction $T_1$ returns to the application at 10 am, a transaction $T_2$ issued at 10.01 am to a different node must see $T_1$’s writes. This is called strict serializability. This stricter guarantee is readily available in WiSer, provided the application waits until its original transaction reaches SERIALIZE stage. By definition, any successful transaction (and query) submitted after that point must see all serialized changes, so will see the original transaction. Of course, if the original transaction ran on a poorly connected node and the application timed-out before SERIALIZE, it has to keep polling to check if its transaction did get serialized successfully.

2.2 Visibility and Efficient Publish

Once a transaction has been serialized, it needs to be visible to any subsequent query. Technically, the query can read SerializeFrontiers table, accordingly contact individual nodes for segments of their local log, and thus scan the entire history of database deltas. This, with lookups (antijoins) into Rollbacks and ConstraintFailures tables, gives full visibility to the serialized state.

But OLTP applications have very strict latency requirements, and timeouts are perceived as failures. Scanning the entire log is prohibitively slow. This is why every node runs a Publisher that scans the recent append to the log at that node and converts them to a query-efficient form. This involves three pieces:
Organizing data by tables: 

- WiSer separates the deltas by table (recall that every modification is an upset), and appends them to per-table blocks, still in Parquet format. Every row is timestamped with a **serializer sequence number (SSN)**, which is an identifier for each transaction in the serial order. The SSN is used by queries to antijoin with **ConstraintFailures** table to remove entries from transactions that failed aggregation constraints (checking constraints happens after publish).
- Removing deltas from conflict rollbacks: WiSer also performs an antijoin with Rollbacks table to remove all rows from transactions that hit read-write conflicts.
- Updating indexes: OLTP queries often do highly selective ("point") lookups that need indexes to be efficient, so WiSer has to update these as well. WiSer partitions the primary key index the same way as the data, so that update is also a local operation. We use an LSM-based indexing structure so that incremental inserts are efficient and incur low latency even on large tables. Maintaining secondary indexes in WiSer is a more expensive, shuffle-like data movement, so we expect to maintain only a limited number of secondary indexes at **PUBLISH** stage. WiSer implements the HERMIT technique to reduce the storage consumption caused by secondary indexes.

Experimentally we find **PUBLISH** stage to be the throughput bottleneck for WiSer, and so **PUBLISH** is done in parallel, across all nodes. This means publication into tables can happen **out of serial order**. So we maintain a **PublishFrontiers** table, whose schema is identical to **SerializeFrontiers**. Each row of this table contains a node ID and an SSN up to which log entries from that node have been published. Queries access published tables (and primary indexes) to access all content up to the **minimum** publication frontier (across all nodes). Beyond that (until the serialization frontier), queries need to scan logs. Most transactions avoid this scan by waiting until the publication frontier catches up to their snapshot, which they picked up by reading **SerializeFrontiers** at transaction start. In this way, they only need to scan logs if they have very tight latency requirements.

One thing that **PUBLISH** does not do is table repartitioning: all the data ingested at one node stays at that node. Clients can carefully route transactions to respect any desired clustering properties. For example, they may send a transaction that inserts a fact row to a node based upon the hash of the primary key or based on geography. **PUBLISH** simply converts logs to table format, local to each node.

### 2.3 Node Departures and Re-Joins

WiSer has a **Nodes** table that contains the list of nodes. Thus node join events get serialized just like other database modifications.
Node departures from the network are not supported directly. If a node becomes unresponsive, its peers cannot directly evict that node, because all the transactions that reached Promise on that node must still be serialized. This is handled by the node replicas. When the node re-joins the cluster, it gets a new node ID.

### 3. CONFLICT RESOLUTION

Conflicts in WiSer are resolved after we have determined a serial order. Thus the resolution problem is simple: the reads in a transaction **X** saw a snapshot that is earlier than where its delta fits in the serial order. **X**’s delta includes **X**’s read-set and **X**’s snapshot (seen SSN). So the **CONFLICTRESOLVER** needs to traverse the database chain of deltas, starting from the prior resolved point, and verify for each delta whether its read-set could have changed in value due to intervening deltas (between **X**’s snapshot and **X**). If it could have changed, then **X** is added to Rollbacks table. **X** is stored as a transaction ID in Rollbacks table: SSNs are assigned later, at **Publish**, only for the successful transactions.

Note that this process can give false positives – excess rollbacks, for two reasons. First, read-sets and write-sets are tracked approximately as we describe below. Second, in our data structure it is expensive to remove the writes of a rolled-back transaction, so a transaction can roll back due to conflicts with another one, which rolled-back for other reasons. This behavior is similar to the way deadlocks are typically handled, and applications are used to retrying transactions upon such roll-backs.

### 3.1 Representation of Read Sets

WiSer represents read-sets as entries in a ReadWriteSet table. This “all state is in tables” principle means that a transaction’s delta naturally incorporates its read-set as well. Queries are allowed to selectively read data by specifying conjunctive equality predicates on one or more columns, or range predicates (for table scan queries, the read set is escalated to the full table). For the common case of equality predicates, we compute a single 64-bit hash value over the key, thus easily handling different data types and multi-column predicates. For range predicates we store the boundaries of the range.

Our current implementation of WiSer unifies the role of **CONFlictRESOLVER** and **SERIALIZER**. Thus each node running transactions replicates ReadWriteSet to the **SERIALIZER** node, along with its regular heartbeats (the one containing LSN ranges of recent additions to its log).

### 3.2 Representation of Writes

To resolve conflicts, the read-set of a transaction **X** needs to be compared against all writes in the serialization order between **X**’s snapshot and **X**. Recall that **PUBLISH** timestamps every row with its SSN. Thus this comparison can be done by an index lookup when available, and if not then via a table scan over recent data (WiSer maintains a min-max synopsis on each block for each column, so this is not terribly inefficient).

Still, to get greater efficiency in conflict resolution, the **SERIALIZER** caches recent writes in memory directly. When a transaction **X** is during execution, the read-set of **X** is placed in ReadWriteSet as discussed earlier. In addition, the primary keys of rows written by **X** are also placed in ReadWriteSet. Notice that the reads are represented logically (e.g., as key ranges), whereas writes are enumerated row-by-row. As the **SERIALIZER** receives writes, it maintains a list of hash tables. These hash tables map hash(key) onto the latest SSN when that key was modified (using the same function to map arbitrary composite key values onto a 64-bit hash as we did for the reads).

The **SERIALIZER** streams all read-sets it receives, looking for conflicts in the recent writes. A conflict with a write only if (a) the key-range of the read spans the key of the write, and (b) the SSN of the write is greater than the snapshot of **X**.

These writes are maintained as a linked list of hash tables to facilitate eviction. Each hash table holds writes within a range of SSNs. So periodically we drop hash tables for SSN ranges that are older than most transactions.

### 3.3 Catching Phantoms from Range Predicates

Queries with range predicates are challenging to run serializably because of phantoms. The read-set for such a query is a range of key values, but the DBMS must guard against concurrent inserts of any value in that range. Typically this is done by placing a “next-key” lock on the index tree structure – i.e., a guard on the values just beyond the two boundary values of that range [20]. The trouble with
this approach is that it is efficient only when the index is memory-
resident. The space taken to check for conflicts is proportional to
the database size.

WiSer handles conflicts due to range predicates very similarly
to the case of equality predicates. When there is a range index
defined on a column, recent writes are cached at the SERIALIZER
as an in-memory tree of column values (not hashes). This is also
a list of trees, organized by ranges of SSNs, much like the hash tables.
Conflict detection is done by probing read-sets into these trees,
and checking any matching writes for SSNs beyond the snapshot of
the read-set.

4. CONSTRAINT RESOLUTION

We have now built up enough machinery to discuss how aggrega-
tion constraints are checked. These constraints are checked after
conflict resolution and after publish, who has stamped each row
with the SSN (its position in the serial order).

Logically, constraint resolution is a walk through the published
data in SSN order, from the prior resolved point all the way up to
the latest published point. The CONSTRAINTRESOLVER maintains
a running aggregate, and we only support incrementally computable,
algebraic aggregation constraints. The constraint is specified as a
predicate on the result of the aggregation. In the online shopping
application, the aggregation is to compute the sum of quantities
grouped by product IDs, and the predicate is to check whether the
aggregated result is larger than or equal to 0.

The CONSTRAINTRESOLVER can run queries repeatedly to up-
date the running aggregate, each time selecting data in the range
from the prior resolved SSN to a value t, where t is gradually in-
creased until it hits a constraint violation.

But every transaction creates a new SSN, so this approach is
far too slow. We want constraints to be evaluated on batches of
transactions, corresponding to the batches that were serialized from
each node. The challenge is that aggregates are generally not mono-
tonic: just because aggregation constraints are not violated after
incorporating an entire batch of transactions, we cannot assert that
the constraints were not violated at some intermediate point (for
example, a bunch of customer orders might have depleted inventory
to be negative, and then a restocking transaction might have been
serialized).

A second design goal is to do parallel constraint resolution, in
two different forms:

- Source parallelism: We want the bulk of aggregation to be
  performed at the nodes running the OTLP transactions, that reduces
data movement and also gives significant parallelism.

- Resolver parallelism: The running aggregation state for con-
  straint enforcement can be large: for example, it is the running
  inventory of each product in a retail application. So we want indi-
  vidual resolvers to focus on subsets of products.

4.1 Partial Constraint Evaluation

WiSer supports partial constraint evaluation to reduce computa-
tion complexity.

Let us consider a setting with three nodes, each with multi-
ple batches of transactions. The CONSTRAINTRESOLVER repeat-
edly picks up one batch from each node to resolve (recall that
SerializeFrontiers lists the batches). It first pushes down to
each node a partial aggregation query, which is derived from the
constraint (e.g., compute the sum of quantities grouped by product
IDs, in the online shopping application).

Next, the CONSTRAINTRESOLVER uses the partial aggregation
results from each node to compute prefix sums (per the serial order)
for the value of the aggregate at the start of each batch. If the prefix
sum after any batch violates the constraint, the CONSTRAINT-
RESOLVER reduces that batch’s contribution to be one that makes the
constraint just satisfied. For example, if the prefix sum before and
after a batch is 12 and -5, respectively, then the CONSTRAINT-
RESOLVER artificially inflates the prefix sum after that batch to 0.
This is not to abort the whole batch; instead, it is only to avoid polluting
the prefix sums for other batches. In effect, the CONSTRAINT-
RESOLVER is delegating to the nodes the responsibility of keeping
the constraint satisfied by rolling back any transactions that cause
violations.

After that, the CONSTRAINTRESOLVER communicates these
prefix sums to each node, and asks it to re-evaluate the aggregation
query, except this time with a modified aggregation function. This
new query runs the same group-by, but initializes its aggregation
hash table with the prefix sum from the CONSTRAINTRESOLVER.
The aggregation scans the input in SSN order (this does not involve
an extra sort, because the rows were published in that order). As
the query maintains the running aggregate it continually applies the
constraint (e.g., inventories should stay positive), and surgically
identifies and excludes individual transactions that cause constraint
violation.

As a side-effect of this query, each node now adds rows to the
ConstraintFailures table – these rows contain just the SSN of the
constraint-violating transactions.

Notice that excluding these transactions will cause the prefix
sums to change, but this is not communicated back to the CON-
STRAINTRESOLVER. After all, removing transactions that caused
constraints to be violated only moves the running aggregate in a
beneficial direction, so it is safe for other nodes (checking other
batches) to ignore this effect.

Constraint resolution is the last stage of transaction processing.
Queries do antijoins with ConstraintFailures to avoid seeing
changes from transactions that fail constraint resolution. Transactions
that reached PROMISE but failed conflict resolution or con-
straint resolution are aborted transactions.

5. PERFORMANCE EVALUATION

WiSer is the next version of the Wildfire research prototype [5]. It
is a HTAP DBMS that currently supports ingest, via Apache Spark
and via a custom API; queries, directly via a SQL dialect, as well
as ones pushed-down from Spark, and now multi-statement trans-
actions. The last feature is the focus of this section. Transactional
changes are replicated and hardened to a quorum of nodes. Tables
can optionally have a primary key index on one or more columns,
and this index supports a mix of equality and range predicates. All
tables and logs are organized as Apache Parquet blocks. We use the
parquet-cpp writer.

All experiments in this section are run in a Docker environment,
on a machine cluster with 28 nodes and 1.6 TB DRAMs in total.
Writes are hardened to PCI-e attached SSDs with fsync. We execute
one transaction thread in each node; the background operations
such as serialization, conflict resolution, and publication, are all
implemented on separate nodes from those running transactions.

5.1 Throughput with No Conflicts

Our first experiment investigates the basic transaction processing
throughput in WiSer, without considering conflicts or rollbacks.
Transactions are run up to SERIALIZE. CONFLICTRESOLVER does
nothing, because there are no conflicts. We do not run PUBLISH for
this experiment.

Our workload consists of just the NewOrder transaction from the
online shopping application discussed in Section 1 run repeatedly,
with no think-time, and no aggregation constraints. Products table has 10,000 items, with an index on productId column. Each transaction looks up the current price of 10 products and inserts an order containing those 10 products into Orders table. The values in OrderId column are generated as sequential numbers, and there are no price updates, so in this experiment there are no conflicts.

Figure 4 shows the transaction throughput we obtain with this simple workload, and the scaling. Notice that at 24 nodes we are inserting 3.5 million rows/s into Orders table, and correspondingly doing 3.5 million lookups/s into the index on Products table. In this setting, the X-axis is the number of nodes running the NewOrder transaction. We have a separate node running the SERIALIZER and CONFLICTRESOLVER. Notice also that we scale well; our system has pretty much no contention, except for the log writes (which is implemented as a lock-free linked list of blocks). The same plot also shows what happens if we turn off serialization: the effect is negligible.

Note that each transaction implicitly also generates 10 reads and 10 writes, which are tracked and sent to the SERIALIZER. From profiling we verify that most of the time goes into this tracking of read-write sets.

5.2 Throughput with Conflicts

Next, we add conflicts, by introducing price updates for the products. We run a UpdatePrice transaction every 100 ms, and that updates the price for a certain number of randomly chosen products. This introduces read-write conflicts because the price that a NewOrder transaction looks up might become stale by the time it goes to SERIALIZE stage.

Figure 5 shows the NewOrder transaction throughput, for both issued and committed transactions, as a function of the number of nodes executing NewOrder transactions. Recall that there is a separate node for serialization and conflict resolution, and the UpdatePrice transactions are issued in another thread. We plot this at two price update volumes: where we update 10 prices in each UpdatePrice transaction, and where we update 20 prices in each UpdatePrice transaction.

Notice that the issued and committed throughputs are almost identical. We measure about 1% rollback rate when we update 10 prices at a time, and 2% when we update 20 prices at a time.

There are 10,000 products, and each transaction is looking up 10 random products. Our SERIALIZER picks up batches of transactions to serialize roughly every 100 ms. Thus, if the SERIALIZER keeps up perfectly with the transaction rate, each NewOrder transaction will overlap with about one UpdatePrice transaction (in the sense that the snapshot that the NewOrder sees will be behind its serialization point by about one UpdatePrice transaction).

This expected behavior matches the rollback rate well. When an UpdatePrice updates 10 prices, a given product has a 10/10000 or 0.001 chance of getting updated. So the chance that a transaction will commit is \((1 - 0.001)^{10}\), or about 0.99. Likewise, when an UpdatePrice updates 20 prices, the chance that a transaction will commit is \((1 - 0.002)^{10}\), or about 0.98, which matches our observed rollback rates.

5.3 End-to-End Throughput

Our next experiment looks at the end-to-end transaction throughput. We add the PUBLISHER, and hence what we measure is the throughput of published transactions. We have one PUBLISHER per (logical) node, so we need as many PUBLISHER nodes as NewOrder nodes. Thus this experiment scales only up to 12 nodes. (12 nodes for NewOrder, one for UpdatePrice, one for SERIALIZER and CONFLICTRESOLVER, 12 for PUBLISHER). Figure 6 shows the effect of adding publish on the system throughput. There is about a 70% hit on throughput, and our scaling is affected. PUBLISHER has to write data to Parquet blocks, and we find this to be a bottleneck.

5.4 Latency Analysis

Our final experiment looks at transaction latency. We rerun the full experiment, with NewOrder and UpdatePrice transactions, as well as SERIALIZER and CONFLICTRESOLVER, and PUBLISHER. This time we measure the average NewOrder transaction latency, until PUBLISH and SERIALIZE stages. Figure 7 plots this latency as a function of the number of NewOrder threads. Observe that latency to serialize is in the tens of ms, and this is the minimum latency until a query can access the transaction’s changes (by reading the
6. RELATED WORK

WiSer builds on and borrows hugely from the large volume of literature on transaction processing, where both shared-nothing and shared-storage models have been heavily studied \[9, 11\].

After many efforts on NoSQL databases that downplayed the importance of consistency, the last decade has seen a renewed interest in systems that provide strongly-consistent transactions over large-scale distributed systems, across partitions and replicas. One type of distributed DBMS, represented by IBM’s Spinnaker \[27\] and Google’s Spanner \[10\], adopts 2PC to coordinate distributed transactions and exploits consensus protocols like Paxos \[13\] or Raft \[22\] to synchronize replicas during transaction execution. In Spanner, write transactions consult a lock server, resembling classical shared-nothing DBMSs.

Another approach, taken by systems like Calvin \[29\] and FaunaDB \[14\], is to use a deterministic execution model. These DBMSs coordinate transactions within a pre-processing phase. In this phase, the DBMS must extract the write-set from the to-be-executed transaction, and accordingly choose a serial order. WiSer, like Calvin, tries to come up with a serial order, and then efficiently do post-processing on the deterministic state. But unlike Calvin, WiSer does not pre-process transactions and thus can support transactions that are not pre-analyzable stored procedures.

Many modern DBMSs attempt to do analytics with transactions \[24\]. HyPer \[17\] is a modern main-memory DBMS that targets efficiently supporting OLTP and OLAP workloads in the same memory space. It leverages multi-version concurrency control (MVCC) and precision locks to achieve fully serializable transaction processing without blocking on-the-fly analytical queries. Similar to HyPer, Peloton \[25\] also leverages MVCC \[30\] to isolate on-line transactions from analytical queries, but it also leverages hybrid storage layout to better support different types of queries \[3\]. Focusing on modern hardware, Appuswamy et al. \[2\] studied the case for developing HTAP DBMSs by exploiting the power of both CPUs and GPUs. A number of commercial systems also claim HTAP features, such as SAP HANA \[13\] and IBM DB2 Event Store. SQL Server introduced their enhancements for hybrid workloads in 2016 \[19\]. By improving support for column-store indexes, SQL Server can enable real-time analytics concurrently with transactional processing.

DBMSs have long leveraged sharding techniques to store data across multiple nodes. Sharding simplifies scaling, but does bring in distributed transactions, which can significantly degrade system scalability. Harding et al. \[15\] performed an experimental evaluation of distributed concurrency control protocols. Some systems use transaction chopping to serialize transactions \[34\]; \[21\]; \[31\]. H-Store/VoltDB \[16, 28\] uses partition-level locks to reduce the overhead of distributed operations. Other systems try to avoid data partitioning. For example, Hyder \[6\] exploits SSDs to scale DBMSs in a shared-flash setting, whereas FaRM \[12\] uses RDMA and non-volatile DRAM to avoid user-level partitioning.

7. CONCLUSIONS AND FUTURE WORK

The distributed systems community has developed consensus protocols mostly independently from the database community’s struggles with distributed transactions. We have described one approach to exploiting consensus as the underlying mechanism for transaction serialization, and shown it provides two key benefits: higher availability during the application’s interaction with the running transaction (the promise step), and greater scaling and easier enforcement of aggregation constraints after serialization.

WiSer comes at a time when demand for HTAP and real-time analytics is exploding. Yet, OLTP systems are complex and well-entrenched “mission-critical” pieces of applications. So we have an uphill task to penetrate this market, requiring that we massage our ‘promise-serialize’ step to fit with existing applications, equal or surpass the plethora of must-have OLTP features (meeting existing isolation semantics, handling referential integrity, etc.), and demonstrate value from pulling compensation logic out of applications.

WiSer’s design of putting all state into tables, and having a tiny consensus piece, may also help in Byzantine settings. Blockchain systems are to some extent bypassing and re-inventing many aspects of classical OLTP. We want to explore whether casting OLTP as a consensus problem will make it easier to provide higher security, timestamping, and non-perturbation guarantees within existing DBMSs.

8. REFERENCES
