Solar: Towards a Shared-Everything Database on Distributed Log-Structured Storage

Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, Huiqi Hu
Background

- **Single-Node In-Memory DBMS**
  - ✓ High xact processing performance
  - ✗ Limited scalability

- **Shared-nothing DBMS**
  - ✓ Scale out via horizontal partitioning
  - ✗ Poor performance w/ distributed xact

- **Shared-everything DBMS**
  - ✓ Scalable storage and xact via fast inter-node communication
  - ✗ Expensive network infrastructure
Design considerations
- General workloads w/ distributed transactions
- Storage scalability
- Commodity machines

Goal: high performance OLTP DBMS w/o assumption on workloads or hardware
Architecture

- **Overview**
  - Several **S-nodes** *(storage & snapshot read)*
  - A **T-node** *(transaction validation/commit & delta read)*
  - Several **P-units** *(business logic processing)*

![Diagram of Architecture](image)

- **Storage Layer**
- **Transaction Layer**
- **Processing Layer**
Architecture

- **S-nodes**
  - Distributed storage engine
  - Role: storing a consistent database snapshot (SSTable)
  - Feature: supporting scalable data storage

<table>
<thead>
<tr>
<th>Tablet 1</th>
<th>Tablet 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
</tr>
<tr>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>6</td>
<td>6.0</td>
</tr>
</tbody>
</table>
Architecture

- **T-node**
  - In-memory transaction engine
  - Role: managing newly committed data since the last snapshot (Memtable)
  - Feature: servicing performant transaction writes

### Tablet 1

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2.0</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3.0</td>
<td>30</td>
</tr>
</tbody>
</table>

### Tablet 2

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4.0</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5.0</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6.0</td>
<td>60</td>
</tr>
</tbody>
</table>
Architecture

- **P-units**
  - Distributed query processing engine
  - Role: SQL, stored procedure, query processing, remote data access
  - Feature: scalable computation power

<table>
<thead>
<tr>
<th>Tablet 1</th>
<th>Tablet 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
</tr>
</tbody>
</table>
LSM-Tree style storage

- **SSTable**
  - A consistent snapshot
  - Data partitioned into tablets (ranges over tables)
  - Tablets replicated over S-nodes

- **Memtable**
  - Newly committed data
  - Stored in memory on T-node
  - Multi-version storage
  - Replicated to backup T-nodes

```
<table>
<thead>
<tr>
<th>Tablet 1</th>
<th>Tablet 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Item Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>
```
Transaction Processing

- Start a transaction
- Read a record
- Process the SQL
- Write a record
- Commit

```
UPDATE Item
SET quantity = quantity - 15
WHERE id = 3;
```

<table>
<thead>
<tr>
<th>Tablet 1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
<td>quantity</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buffer</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>quantity</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S-node</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Item(id=1, quantity=30)</td>
<td>commit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T-node</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>NULL</td>
</tr>
</tbody>
</table>

| 1  | quantity=5 | NULL |
| 3  | quantity=15 | NULL |
Data Compaction

- All data are firstly written into the T-node
- Data compaction moves committed data into S-nodes
  - Does not block on-going and future transactions
Concurrent Control

- Use MVOCC to support Snapshot Isolation (SI)
  - Prevent lost update anomaly
- Data structures on the T-node
  - A timestamp counter (MVCC)
  - Row-level latch (OCC)
- Snapshot Acquisition
- Transaction Validation

Counter: 5

Txn \( t_x \)
read-timestamp: \( rts = 5 \)

Write(key=1, \( col_1 = 3 \))
Recovery

- **Write ahead log**
  - Generate redo log entries
  - Group commit

- **T-node recovery**
  - Replay redo log entries
  - The replay position is determined by the last finished data compaction

- **S-nodes do not lose data**
- **P-units do not store data**
Data Compaction

- Data compaction (DC) starts when the T-node runs out of memory
  - New Memtable $m_1$ to accept transactions after DC initiation
  - Memtable $m_0$ is frozen and merged into SSTable
- Goal: minimize blocking of transaction processing

```plaintext
initiate Memtable \( m_0 \) (SSTable \( s_0 \))

\[ \text{SSTable } s_0 \rightarrow \text{SSTable } s_1 \]

rw/validate Wait for rw/validate
ro/validate

\( t_x \)
\( t_y \)
\( t_z \)

timeline
```

- Wait for \( t_{dc} \)
Remote Data Access Optimization

- Shared-Everything architecture
  - Latency bounded by remote data access between
    - P-unit and T-node
    - P-unit and S-node
  - Reducing remote data access cost
    - => more concurrent transactions
    - => higher throughput
Local SSTable Cache

- Build SSTable Cache on P-unit
  - SSTable is immutable
  - P-unit examines its local cache before communicating with S-nodes
Asynchronous Bit Array

- Empty reads on the T-node
  - The T-node stores a small part of data
  - Reading non-existing data items results in useless empty reads

<table>
<thead>
<tr>
<th>Tablet 1</th>
<th>Tablet 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>id  price quantity</td>
<td>id  price quantity</td>
</tr>
<tr>
<td>1   1.0 10</td>
<td>4   4.0 40</td>
</tr>
<tr>
<td>2   2.0 20</td>
<td>5   5.0 50</td>
</tr>
<tr>
<td>3   3.0 30</td>
<td>6   6.0 60</td>
</tr>
</tbody>
</table>
Asynchronous Bit Array

- **Asynchronous Bit Array**
  - Encode whether any row in Tablet \( x \) has its column \( y \) modified
  - Periodically synchronized to P-units
    - False positive => empty read (corrected after the first access)
    - False negative => validating empty reads and retry

<table>
<thead>
<tr>
<th>Tablet 1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
<td>quantity</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tablet 2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
<td>quantity</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>5.0</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>6.0</td>
<td>60</td>
</tr>
</tbody>
</table>

Any data in the T-node

<table>
<thead>
<tr>
<th></th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablet 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Tablet 2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Transaction Compilation

- Model a transaction as a directed acyclic graph
- Move reads to start if possible

\[
v_{\text{balance}} = r2.\text{balance}
\]
\[
v_{\text{price}} = r1.\text{price}
\]
\[
v_{\text{balance}} - = v_{\text{price}}
\]
Buffer Write (\textbf{Cust}, r2, balance = v_{\text{balance}})

Start

Memtable Read (\textbf{Item}, r1)

Memtable Read (\textbf{Cust}, r2)

SSTable Read (\textbf{Item}, r1)

SSTable Read (\textbf{Cust}, r2)

Memtable Read (\textbf{Item}, r1)

SSTable Read (\textbf{Cust}, r2)

Memtable Read (\textbf{Cust}, r2)

SSTable Read (\textbf{Item}, r1)

SSTable Read (\textbf{Cust}, r2)

v_{\text{balance}} = r2.\text{balance}

v_{\text{price}} = r1.\text{price}

v_{\text{balance}} - = v_{\text{price}}

Buffer Write (\textbf{Cust}, r2, balance = v_{\text{balance}})

Commit
Transaction Compilation

- Group T-node access

```
Memtable Read (Item, r1)  Memtable Read (Cust, r2)
SSTable Read (Item, r1)  SSTable Read (Cust, r2)

v_balance = r2.balance  v_price = r1.price

v_balance -= v_price

Buffer Write (Cust, r2, balance = v_balance)
```
Pre-execute S-node access

Memtable Read (Item, r1) → SSTable Read (Item, r1) → v_balance = r2.balance → v_balance -= v_price → Buffer Write (Cust, r2, balance = v_balance)

Memtable Read (Cust, r2) → SSTable Read (Cust, r2) → v_price = r1.price → v_price = r1.price → Buffer Write (Cust, r2, balance = v_balance)
Experiment

- **Setting**
  - CPU: 2.4G Hz 16-Core
  - Memory: 64GB
  - 10 servers
  - Connected by 1 Gigabits Network

- **Benchmark**: TPC-C

- **Systems**
  - Workload: TPC-C
  - MySQL Cluster
  - VoltDB
  - Tell

---

**Scalability**

- Throughput (tps)

  - Solar
  - Tell-1G
  - MySQL-Cluster
  - VoltDB
  - Tell-10G
  - Oceanbase

- Number of Servers (#)

**Cross-Partition Transactions**

- Throughput (tps)

  - Solar
  - Tell-1G
  - MySQL-Cluster
  - VoltDB
  - Tell-10G
  - Oceanbase

- Cross-Warehouse Transaction Ratio (%)

22
Experiment

- Data compaction
  - Normal
  - Compaction

- System recovery
  - Kill Snode
  - Restart Snode
  - Kill Tnode
  - Restart Tnode

- Remote data access optimization
Summary

- Solar
  - A shared-everything OLTP DBMS on Commodity hardware
    - High performance transaction processing
    - Scalable data storage capacity
  - Several novel optimization to improve performance
  - Empirical evaluation shows great performance and scalability
- Group T-node access
  - Normal execution issues T-node access one-by-one
  - Try to batch multiple T-node communications together

```
S-nodes \rightarrow P-unit \rightarrow T-node

read(r_1.static) \rightarrow \text{local process} \rightarrow \text{start} \rightarrow \text{read}(r_1\.delta) \rightarrow \text{commit}

read(r_2.static) \rightarrow \text{local process} \rightarrow \text{read}(r_2\.delta) \rightarrow \text{commit}
```

```
S-nodes \rightarrow P-unit \rightarrow T-node

\text{start} \rightarrow \text{read}(r_1\.delta) \rightarrow \text{commit}

\text{read}(r_2\.delta) \rightarrow \text{commit}

\text{start} \rightarrow \text{read}(r_1\.delta) \rightarrow \text{read}(r_2\.delta) \rightarrow \text{commit}

\text{read}(r_1\.static) \rightarrow \text{local process} \rightarrow \text{read}(r_2\.static) \rightarrow \text{local process} \rightarrow \text{commit}
```
Transaction Compilation

- Pre-execute S-node access
  - Normal execution issues S-node access after transaction is started
  - Try to pre-execute S-node reads
Architecture

- Design considerations
  - A shared-everything architecture
    - 2-Layer LSM-Tree style storage
    - Decouple computation from storage
  - High performance in-memory transaction processing
    - MVOCC, combining the OCC and the MVCC
    - A non-blocking data compaction algorithm
  - Fine-grained remote data access
    - Data cache
    - Asynchronous bit array
    - Transaction compilation

Goal: high performance OLTP DBMS without assuming a partitionable workload or advanced hardwares
Architecture

- Overview
Log-Structured write

Write Request

P-unit

T-node

S-node

Write Request

P-unit

Write Request

P-unit
Architecture

- Log-Structured write
Architecture

- **S-nodes**
  - Distributed storage engine
  - Role: storing a consistent database snapshot (SSTable)
  - Feature: supporting scalable data storage
Architecture

- **T-node**
  - In-memory transaction engine
  - Role: managing the rest recently committed data (Memtable)
  - Feature: providing performant transactional writes

![Diagram of multi-node architecture with T-node and S-node, illustrating compaction and in-memory storage.]
**Architecture**

- **P-units**
  - Distributed query processing engine
  - Role: SQL, stored procedure, query processing, remote data access
  - Feature: providing scalable computation power

```
P-node
  Application Logic
  Compiler
  Bit Array
  Storage Interface

S-node
  Tablet
  Tablet
  Tablet
  Disk-based storage

T-node (Replica)
  Bit Array
  Indexes
  Txn Log
  In-memory storage
```

```
P-node
  Application Logic
  Compiler
  Bit Array
  Cache
  Storage Interface

P-node
  Application Logic
  Compiler
  Bit Array
  Cache
  Storage Interface

P-node
  Application Logic
  Compiler
  Bit Array
  Cache
  Storage Interface
```
LSM-Tree style storage

- **SSTable**
  - A consistent snapshot
  - Partitioned into tablets
  - Replicated over S-nodes

- **Memtable**
  - Newly committed data
  - In-memory stored in the T-node
  - Multiple version storage
  - Replicated to backup T-nodes

<table>
<thead>
<tr>
<th>Tablet 1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
<td>quantity</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>30</td>
</tr>
</tbody>
</table>

S-node

<table>
<thead>
<tr>
<th>Tablet 2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
<td>quantity</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>5.0</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>6.0</td>
<td>60</td>
</tr>
</tbody>
</table>

S-node

<table>
<thead>
<tr>
<th>Item Table</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>price</td>
<td>quantity</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5.0</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6.0</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
Read & Writes

- **Read**
  - read and merge versions from both T-node and one of S-node

- **Write**
  - directly write into the T-node
Transaction Processing

- P-unit execute transactions
  - Start a transaction
  - Fetch records from remote
  - Execute user-defined logics
  - Buffer data writes
  - Commit the transaction
Background

- Single-Node In-Memory DBMS
  - Hekaton, HyPer
  - Features
    - No disk I/O during transaction processing (In-memory storage)
    - Transaction compilation
    - Lightweight concurrency control (OCC, MVCC, determinism)
    - Simplified write-ahead logging
    - Very high performance transaction processing
  - Limitations
    - Database size should be smaller than memory capacity
Background

- **Shared-Nothing DBMS**
  - VoltDB/HStore, Spanner
  - **Features**
    - Use horizontal partition
    - Reply on two phase commit
    - Scalable transaction processing and storage
  - **Limitations**
    - Partitionable workload
    - Low percentage of distributed transactions
Background

- **Shared-Everything DBMS**
  - Oracle RAC, Tell
  - **Features**
    - Share data/cache among nodes
    - Rely on fast inter-node communication
    - Scalable transaction processing and storage

- **Limitations**
  - Require advanced network infrastructure
    - InfiniBand switch with 43TB/s, 216 ports costs about $60,000
Transaction Compilation

- Many remote data access between *start* and *commit*
- Group reads to reduce read latency

\[
\begin{align*}
v_{\text{price}} &= \text{Read} (\text{Item, id = 1, price}) ; \\
v_{\text{balance}} &= \text{Read} (\text{Cust, id = 5, balance}) ;
\end{align*}
\]

Start

- **Memtable Read** *(Item, r1)*
- **SSTable Read** *(Item, r1)*
- \( v_{\text{price}} = r1.\text{price} \)

- **Memtable Read** *(Cust, r2)*
- **SSTable Read** *(Cust, r2)*
- \( v_{\text{balance}} = r2.\text{balance} \)
- \( v_{\text{balance}} -= v_{\text{price}} \)

- **Buffer Write** *(Cust, r2, balance = v_{\text{balance}})*

Commit
Reorder ops w/o data dependency does not change semantics

```
Memtable Read (Item, r1)
```

```
SSTable Read (Item, r1)
v_price = r1.price
```

```
Memtable Read (Cust, r2)
```

```
SSTable Read (Cust, r2)
v_balance = r2.balance
```
Transaction Compilation

- Only ops w/ data dependencies cannot be reordered
  - Use the same variable, and one is write (identify by variable name)
  - Use the same record, and one is write (identify by table name)

---

Memtable Read (Item, r1)

```plaintext
v_price = r1.price
```

SSTable Read (Item, r1)

Memtable Read (Cust, r2)

```plaintext
v_balance = r2.balance
```

SSTable Read (Cust, r2)

Read Item Price

Read Customer Balance

Update Customer Balance

Buffer Write (Cust, r2, balance = v_balance)
Data Access During Compaction

- MemTable Read: always read the new MemTable $m_1$
- SSTable Read
  - Merged data ($Tablet 1$): read from $s_1$
  - Merging data ($Tablet 2$): read from $s_0$ and the frozen Memtable $m_0$
Snapshot Isolation During Data Compaction

Classify transactions into three types:

- **Type 1**: start validation before the compaction is initialized
  - validate on $m_0$, write on $m_0$

- **Type 2**: start validation after the compaction is initialized
  - validate on $m_0$ and $m_1$, write on $m_1$

- **Type 3**: starts processing after the compaction is started
  - validate on $m_1$, write on $m_1$
Recovery during Data Compaction (DC)

- Compaction start log entry (CSLE)
  - Persist when the DC is started
  - Acts as a border of redo log entries

- Compaction end log entry (CELE)
  - Persist when the DC is ended
  - Save the position of the CSLE of the DC

- Recovery procedure
  - Read CELE to find the position of CSLE
  - Replay the redo log from CSLE
  - At first, replay data into $m_0$
  - Once CSLE is encountered, repay data into $m_1$
Any data in the T-node

<table>
<thead>
<tr>
<th>Any data</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablet 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Tablet 2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Synchronization & usage**

- Periodically synchronized to P-units
- A P-unit check its local copy to filter useless T-node access

**Diagram:**

- A copy of bit array
- P-unit
- T-node
- Read (id = 1, price = ?)
Asynchronous Bit Array

- **Synchronization & usage**
  - Periodically synchronized to P-units
  - A P-unit check its local copy to filter useless T-node access

Any data in the T-node

<table>
<thead>
<tr>
<th></th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablet 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Tablet 2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Asynchronous Bit Array

- False positive
  - \((row_x, col_y)\) does not exist on the T-node, but the bit array says yes
    - An empty read
  - Reason: bit array maintained at tablet granularity

<table>
<thead>
<tr>
<th>Tablet 1</th>
<th>id</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tablet 1</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tablet 2</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Asynchronous Bit Array

- False negative
  - A bit array copy may fall behind the latest version
  - \((row_x, col_y)\) exists on the T-node, but the bit array says no
  - Transaction re-check all potential empty reads in the validation phase

A copy of bit array

<table>
<thead>
<tr>
<th>T-node</th>
<th>quantity=5</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>quantity=15</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>Quantity=25</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Any data in the T-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Tablet 1</td>
</tr>
<tr>
<td>Tablet 2</td>
</tr>
</tbody>
</table>
Data Compaction

- **Initiate**
  - Create a new Memtable
  - Freeze the current Memtable
- **Handling ongoing transactions**
  - Case 1: validation starts before the compaction is initiated
    - $t_x$ and $t_y$ are allowed to write data into $m_0$
  - Case 2: validation starts after the compaction is initiated
    - $t_z$ will write data into $m_1$ after the data compaction is started
Data Compaction

- **Start**
  - Get compaction timestamp $t_{dc}$ after $t_x$ and $t_y$ abort or obtain commit TS
    - $t_z$ starts validation only after $t_{dc}$ is obtained
  - Start data compaction after $t_x$ and $t_y$ finish abort/commit
    - Create a new SSTable by merging the old one and the frozen Memtable

![Diagram showing the flow of data compaction with timelines and memtables](image-url)
Data Compaction

- **End**
  - Wait until the $s_1$ is fully created
  - Release the old Memtable and SSTable

[Diagram showing timelines for SSTables and Memtables with stages: initiate, start, end]
Concurrency Control

- Data structures on the T-node
  - A timestamp counter (MVCC)
  - Row-level latch (OCC)

- Start
  - Acquire read-timestamp $rts$

- Process
  - Read latest version specified by $rts$

![Diagram showing data structures and transaction read-timestamp]

Counter: 5

- $key=1$
  - $col_1=5$
  - $wts=4$

- $key=2$
  - $col_2=5$
  - $wts=3$

Txn $t_x$
read-timestamp: $rts = 5$
Concurrency Control

- **Commit**
  - Acquire latches for records in the write set
  - Verify there is no newer version
  - Acquire write timestamp \( wts \)
  - Write and release latches

```
key=1
  wts=6
  col_1=2

key=2
  wts=6
  col_2=2
```

Txn \( t_x \)
read-timestamp: \( rts = 5 \)
write-timestamp: \( wts = 6 \)