Distributed Systems

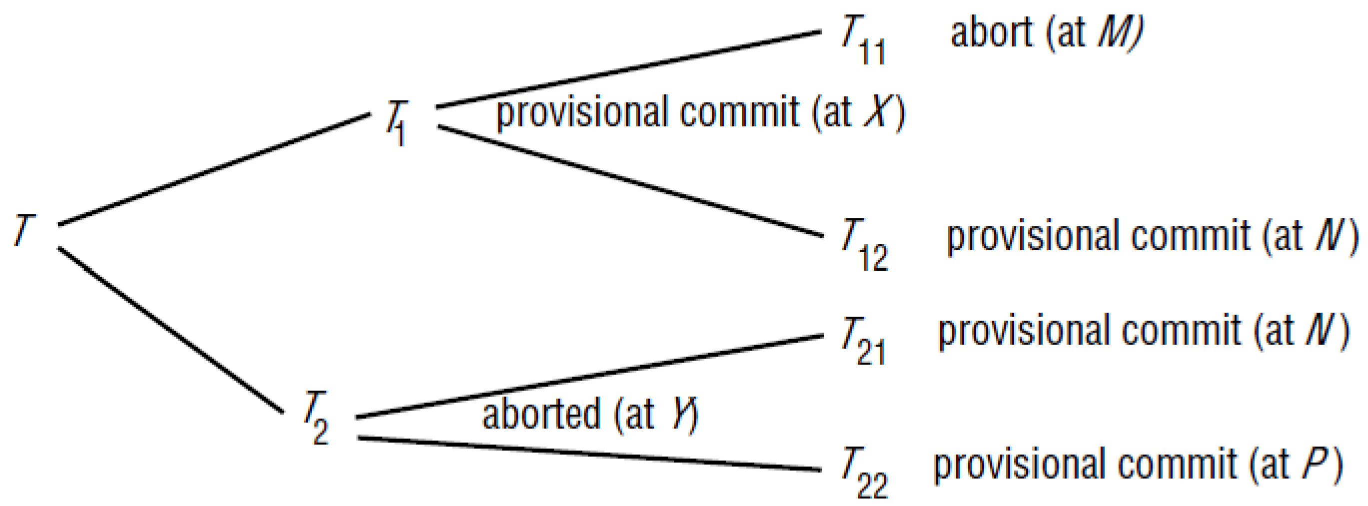
Monsoon 2024

Lecture 17

International Institute of Information Technology

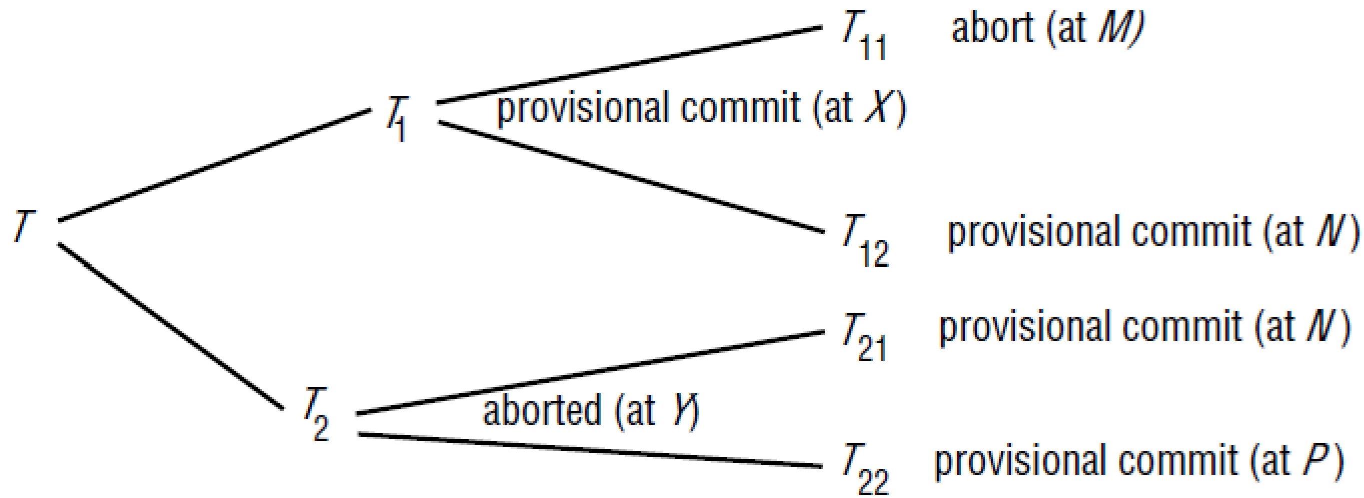
Hyderabad, India

2PC for Nested Transactions



- The outermost transaction in a set of nested transactions is called the *top-level transaction*.
- Transactions other than the top-level transaction are called *subtransactions*.
- For example, *T*11 and *T*12 start after *T*1 and finish before it.

2PC for Nested Transactions



- When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort.
- After all subtransactions have completed, the provisionally committed ones participate in a two-phase commit protocol, in which servers of provisionally committed subtransactions express their intention to commit and those with an aborted ancestor will abort.

Flat Vs. Hierarchical Models

- Hierarchical two-phase commit protocol
  - the two-phase commit protocol becomes a multi-level nested protocol.
  - The coordinator of the top-level transaction communicates with the coordinators of the subtransactions for which it is the immediate parent.
  - It sends *canCommit?* messages to each of the latter, which in turn pass them on to the coordinators of their child transactions (and so on down the tree).
- Each participant collects the replies from its descendants before replying to its parent.

Flat Vs. Hierarchical Models

- Flat 2-Phase Commit Protocol
  - the coordinator of the top-level transaction sends *canCommit?* messages to the coordinators of all of the subtransactions in the provisional commit list.

Other Aspects of Transactions

- Isolation
  - Use locks
  - Be careful about distributed deadlocks
- Recovery
  - Use logs to recover from various states in the 2Phase commit protocol.
  - Possible to extend the logging model to also nested transactions.

Few Optimizations

- Proposed by Mohan et al., there are two particular optimizations to the 2-Phase commit protocol.
- The Presumed Abort and the Presumed Commit are the two modifications.
- Consider completely or partially read-only transactions.
- A transaction is partially read-only if some processes of the transaction do not perform any updates to the data base, while the others do.
- A transaction is (completely) read-only if no process performs any updates.

Few Optimizations

- If a subordinate receives a PREPARE message and it finds that it has not done any updates, then it sends a READ VOTE, releases its locks, and "forgets" the transaction.
- The subordinate writes no log records.
- As far as it is concerned, it does not matter whether the transaction ultimately gets aborted or committed.
- So the subordinate, who is now known to the coordinator to be read-only, does not need to be sent COMMIT/ABORT message by the coordinator.
- There will not be a second phase of the protocol if the coordinator is read-only and gets only READ VOTEs.

From Relational to Non-Relational Models

- Looser schema definition
- Applications written to deal with specific type of data such as documents
- Applications aware of the schema definition as opposed to the data
- Designed to handle distributed, large databases
- Trade offs:
  - No strong support for ad hoc queries but designed for speed and growth of database
    - Query language through the API
  - Relaxation of the ACID properties

RDBMS vs No-SQL

- Elastic scaling
  - RDBMS scale up – bigger load , bigger server
  - NoSQL scale out – distribute data across multiple hosts seamlessly
- DBA Specialists
  - RDMS require highly trained expert to monitor DB
  - NoSQL require less management, automatic repair and simpler data models
- Big Data
  - Huge increase in data capacity and constraints of data volumes at its limits
  - NoSQL designed for big data

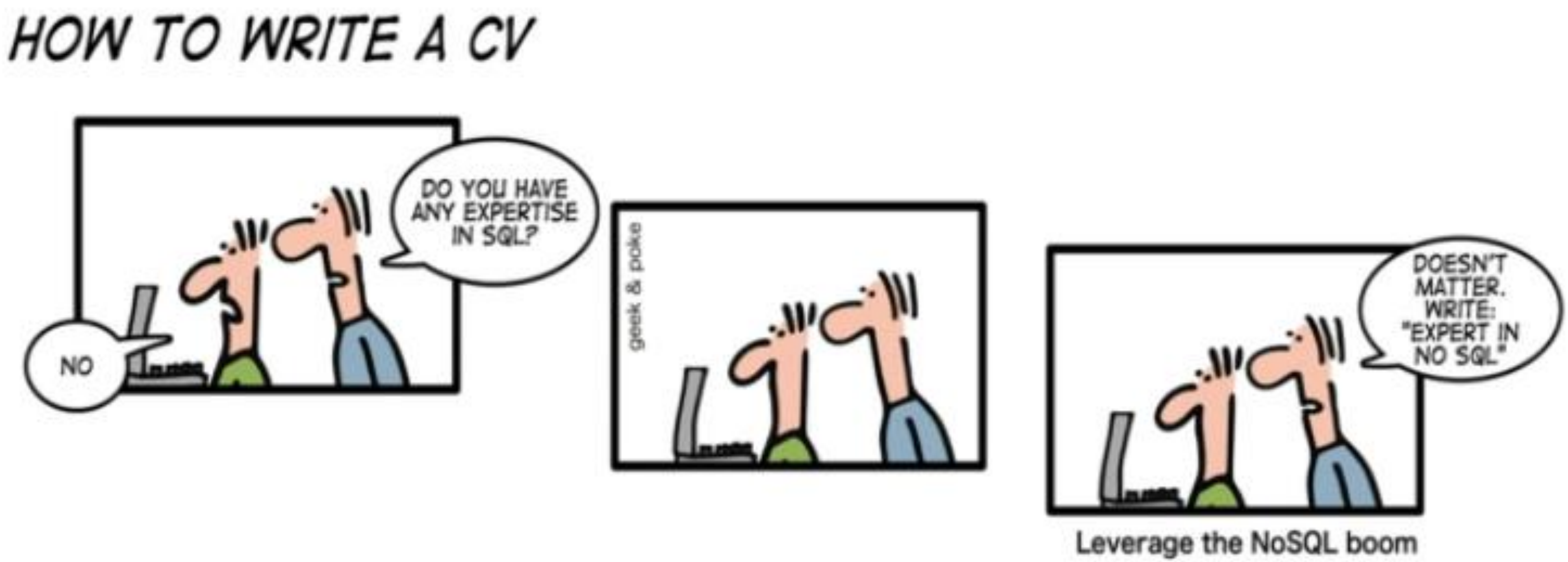
RDBMS vs No-SQL

- Flexible data models
  - Change management to schema for RDBMS have to be carefully managed
  - NoSQL databases more relaxed in structure of data
  - Database schema changes do not have to be managed as one complicated change unit
  - Application already written to address an amorphous schema
- Economics
  - RDMS rely on expensive proprietary servers to manage data
  - No SQL: clusters of cheap commodity servers to manage the data and transaction volumes
  - Cost per gigabyte or transaction/second for NoSQL can be lower than the cost for a RDBMS

RDBMS vs No-SQL Disadvantages

- Support
  - RDBMS vendors provide a high level of support to clients
  - Stellar reputation
  - NoSQL –are open source projects with startups supporting them
  - Reputation not yet established
- Maturity
  - RDBMS a mature, stable and dependable product
  - Also means no longer cutting edge
  - NoSQL are still implementing their basic feature set

RDBMS vs No-SQL Disadvantages



- Lack of Expertise
  - Whole workforce of trained and seasoned RDMS developers
  - Still recruiting developers to the NoSQL camp

RDBMS Vs NoSQL

- RDBMS follow the ACID model for transactions.
- A new name, called BASE, is what NoSQL databases follow.
- BASE stands for Basically Available, Soft-state, Eventually consistent.

Small Detour -- BASE

- Consider web applications that operate at large scale.
- One possibility to scale is the vertical scaling – operate out of the best computer available.
  - Not easy to maintain
- Another popular possibility is the horizontal scaling – operate across many computers.
- One question that then arises is to see how to partition the data. (database).
- Two possibilities:
  - Functional partitioning
  - Sharding

Small Detour

- In functional partitioning, data (tables) are grouped according to function.
- For instance, store user data in one place, e-transaction data in another place, inventory in another, and the like.
- In sharding, the entire database is divided into pieces and each piece is stored in one server.
- Functional partitioning can use sharding as a second level partitioning if the data for one function is too large, for instance.

Small Detour

- Consider a functional partition for the rest of this detour.
- We will see alternatives to ensuring consistency.
- In ACID style semantics, to maintain consistency in the functional partitioning model, one has to use the idea of foreign keys.
- We will see how we can move away from ACID semantics in the case of functional partitioned distributed database.
- Consider a simple schema with just two tables:
  - Users, eTransactions
  - Users table stores data about each user including the total amount transacted
  - eTransactions has one record per transaction performed.

Detour

```
Begin Transaction
    insert into eTransaction(tran_id, seller_id, buyer_id, amount)
    Update User set amount_sold += amount where user_id =
                       seller_id
    Update User set amount_bought += amount where…
End Transaction
```

- We will show that consistency across functional groups is easier to relax than within functional groups.
- Each time an item is sold, a row is added to the transaction table and the counters for the buyer and seller are updated in the Users table.
- Using an ACID-style transaction, the SQL would be as shown.

Detour

- The total bought and sold columns in the user table can be considered a cache of the transaction table.
- Given this, the constraint on consistency could be relaxed.
- The buyer and seller expectations can be set so their running balances do not reflect the result of a transaction immediately.
- This is not uncommon, and in fact people encounter this delay between a transaction and their running balance regularly

Detour

```
Begin transaction
    insert into  eTransaction(id, seller_id, buyer_id, amount)
End transaction
Begin transaction
    update Users set amount_sold = amount_sold + amount
        where user_id = seller_id;
    update Users set amount_bought = amount_bought +
  amount    where user_id = buyer_id;
End Transaction
```

- Notice the two transactions.
- Consistency between the tables is not guaranteed. In fact, a failure between the first and second transaction will result in the user table being permanently inconsistent.

Detour

```
Begin transaction
    insert into  eTransaction(id, seller_id, buyer_id, amount)
    Queue message (Update Users, amount, seller_id, SELL);
    Queue message (Update Users, amount, buyer_id, BUY);
End transaction
For each message in Queue do
    Begin transaction
    if BUY then
        update Users set amount_bought = amount_bought +
        amount    where user_id = buyer_id;
    if SELL …
    End transaction
End-for
```

- One can use queue like message holders to ensure correct updates.
- The modified program looks as follows.

Detour

- Notice the decoupling between the transactions corresponding to the eTranascation table and the Users table.
- Make the queue a persistent object so that updates are not lost.
- There may be one minor problem:
  - The queue has a message, but
  - Due to 2Phase Commit rules, the transaction is not committed!
- Solutions to this problem include
  - Transforming the updates to idempotent updates.
  - Record the list of applied updates separately. Verify each update against this list.

Detour

- The actions above indicate that we are compromising on the Consistency part of ACID semantics a bit.
- The database is always available.
- The state of the database is said to the soft.
- Eventual consistency is ensured because of other mechanisms.
- Therefore, partitioned databases try to support BASE semantics to increase availability.

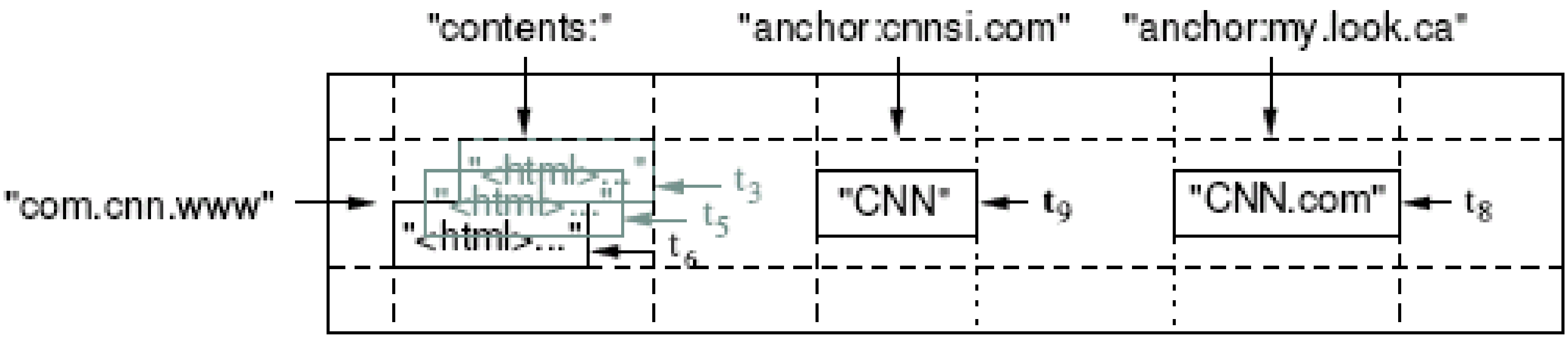
Another Similar Setting

- Consider the Audio Alert box that vendors use.
- Think of the UPI transaction that will have the following style:
  - Debit User Account
  - Credit Vendor Account
  - Insert Audio record
- For the UPI, the third of these should be part of the entire transaction in principle.
- Can decouple this

Another Similar Setting

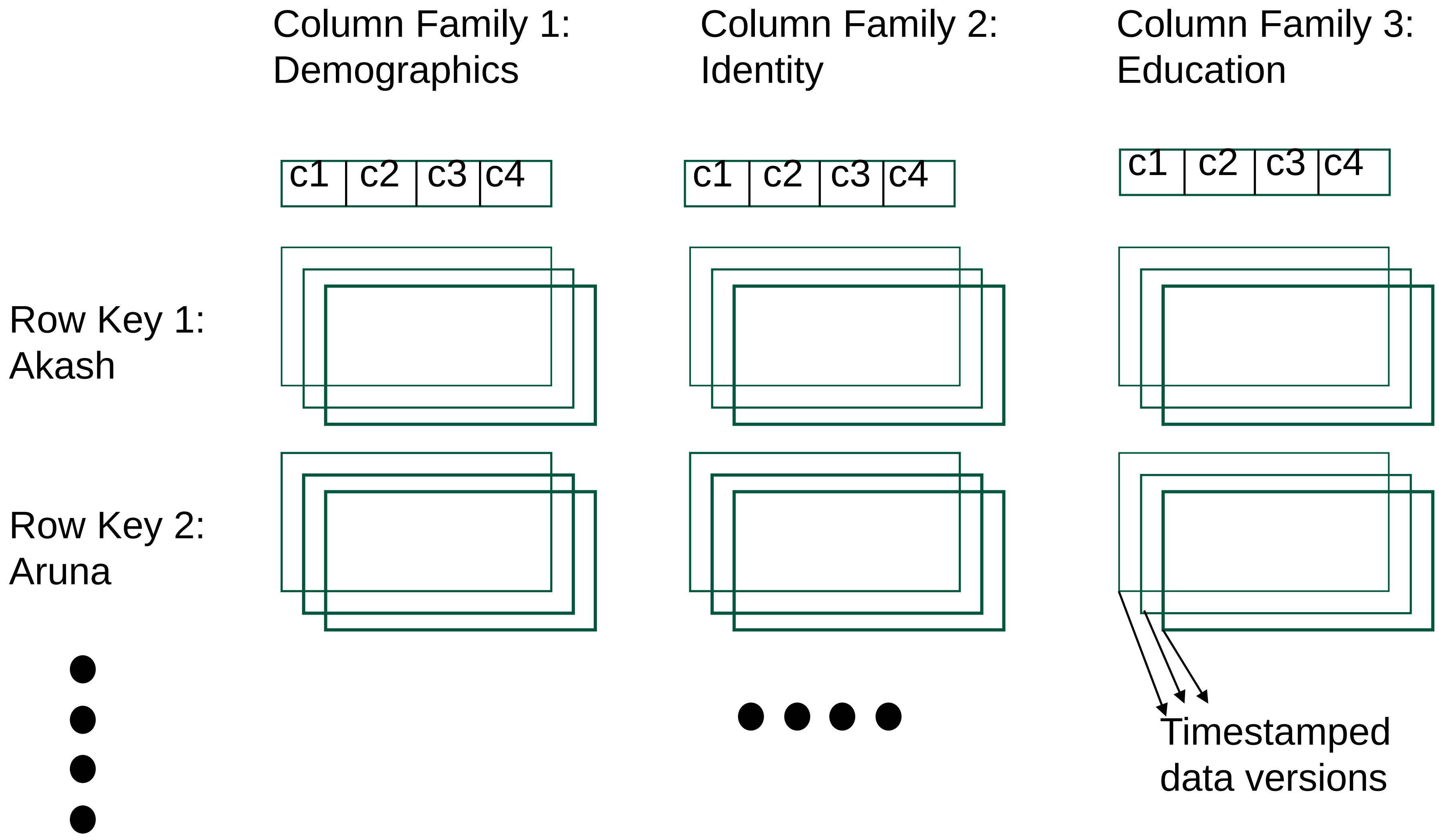
- Imagine that the debit, credit, and the queue entry are three subtransations.
- Imagine a situation where the credit and the debit transactions are ready to commit but the queue entry fails.
- If the parent transaction requires that the queue entry is also equally essential for overall success of transaction, then the entire set of transactions should be aborted.
- This is not desirable since the user faces service disruption for no big reason.

Google BigTable – Data Model

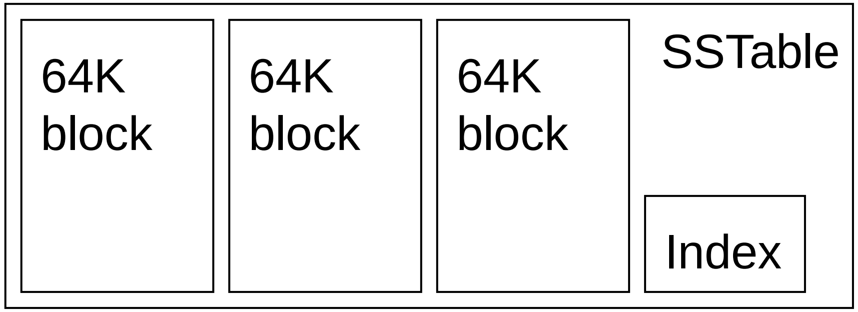


- A Bigtable is a sparse, distributed, persistent multidimensional sorted map.
- <Row, Column, Timestamp> triple for key - lookup, insert, and delete API
- Arbitrary “columns” on a row-by-row basis
  - Column family:qualifier. Family is heavyweight, qualifier lightweight
  - Column-oriented physical store- rows are sparse!
- Does not support a relational model
  - No table-wide integrity constraints
  - No multirow transactions

BigTable Data Model

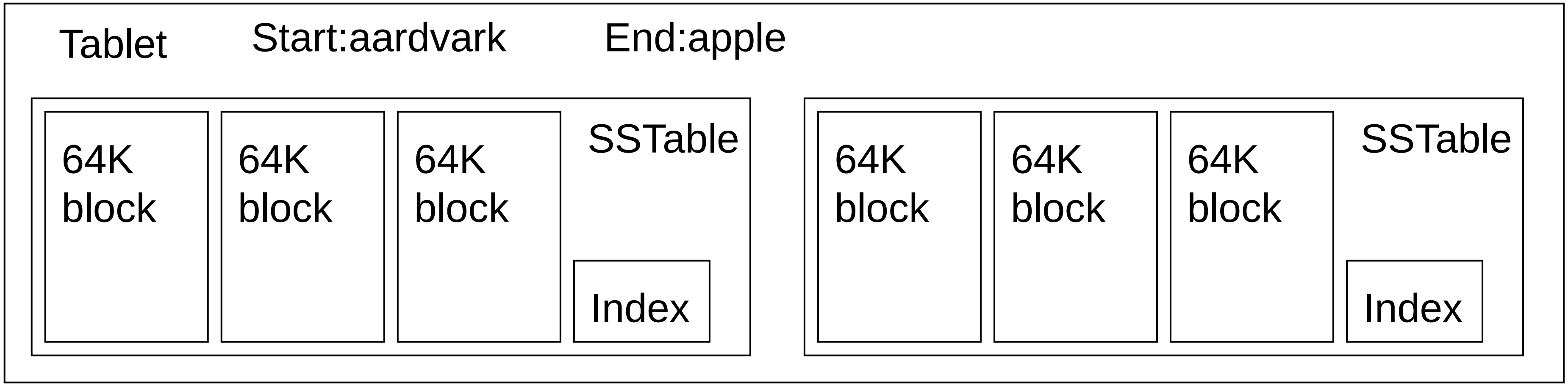


Big Table



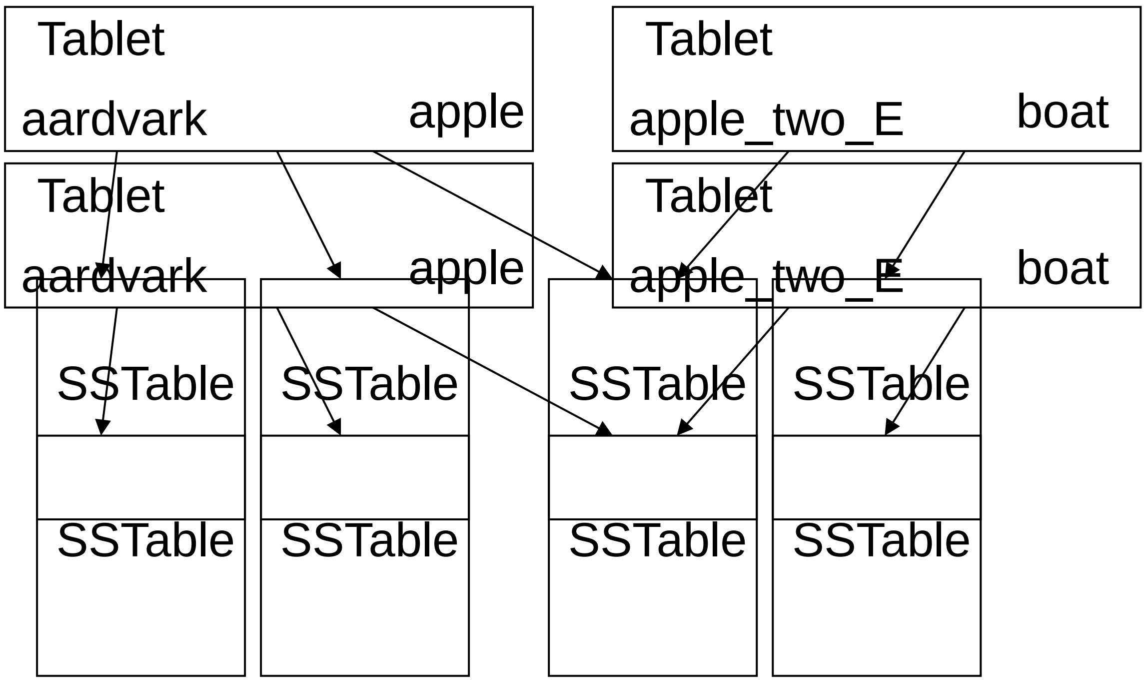
- Building blocks: SSTable
- Immutable, sorted file of key-value pairs
- Chunks of data plus an index
  - Index is of block ranges, not values

BigTable Building Blocks



- Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned
- Tablet: Contains some range of rows of the table
- Built out of multiple SSTables

BigTable Building Blocks



- Multiple tablets make up a table in BigTable
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap

Servers

- Tablet servers manage tablets,
  - Multiple tablets per server.
  - Each tablet lives at only one server
  - Each tablet is 100-200 MB
  - Tablet server splits tablets that get too big
- Master responsible for load balancing and fault tolerance

Master's Tasks

- Use Chubby to monitor health of tablet servers, restart failed servers
  - Tablet server registers itself by getting a lock in a specific chubby directory
    - Chubby gives "lease" on lock, must be renewed periodically
    - Server loses lock if it gets disconnected
  - Master monitors this directory to find which servers exist/are alive
    - If server not contactable/has lost lock, master grabs lock and reassigns tablets
    - GFS replicates data. Prefer to start tablet server on same machine that the data is already at

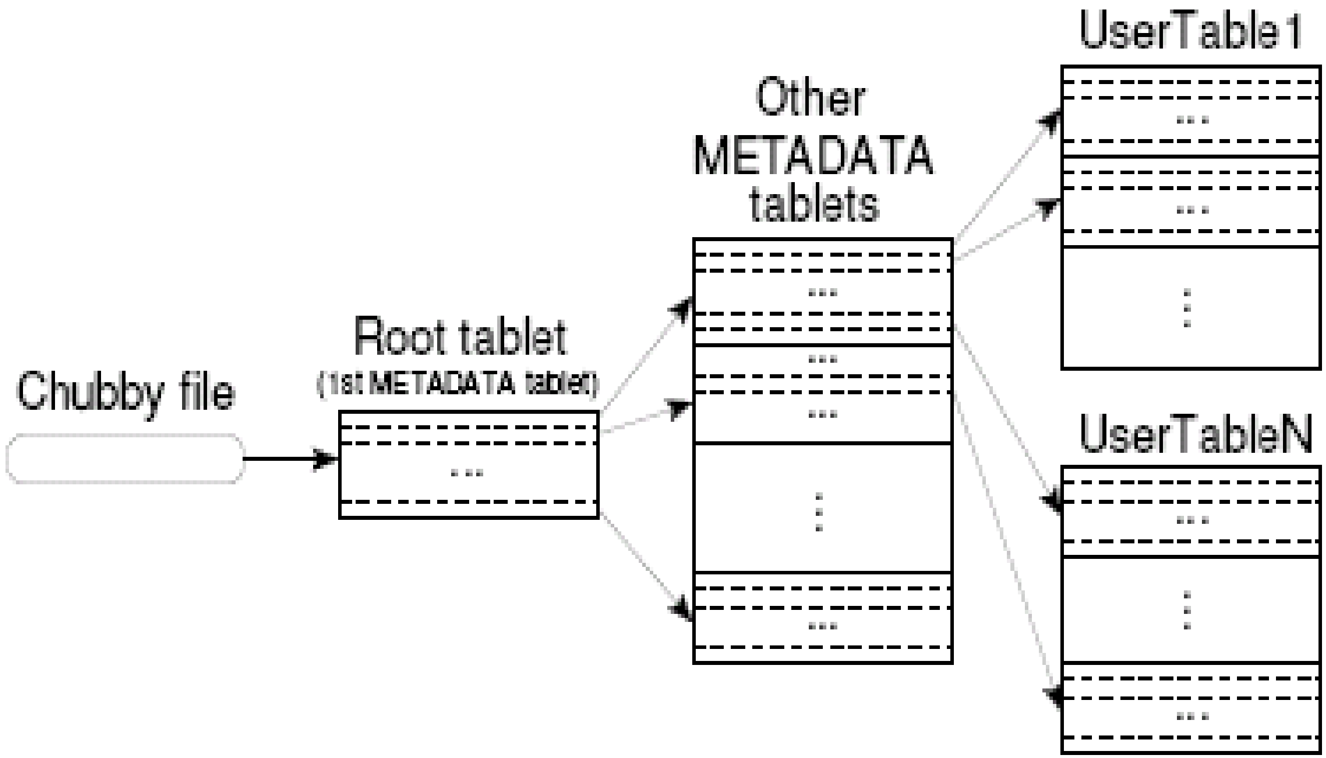
Master's Tasks (Cont)

- When (new) master starts
  - grabs master lock on chubby
    - Ensures only one master at a time
  - Finds live servers (scan chubby directory)
  - Communicates with servers to find assigned tablets
  - Scans metadata table to find all tablets
    - Keeps track of unassigned tablets, assigns them
    - Metadata root from chubby, other metadata tablets assigned before scanning.

Metadata Management

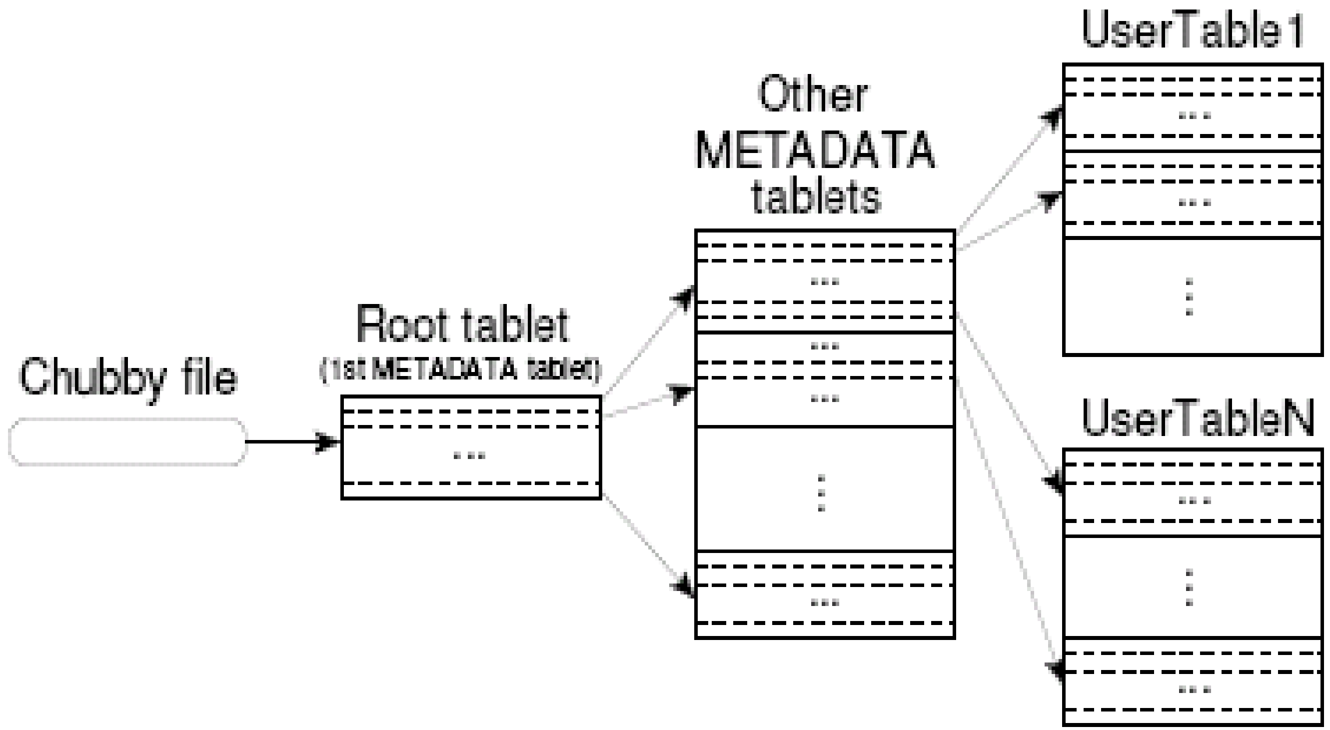
- Master handles
  - table creation and merging of tablet
- Tablet servers directly update metadata on tablet split, then notify master
  - lost notification may be detected lazily by master

Locating a Table/Tablet/SSTable



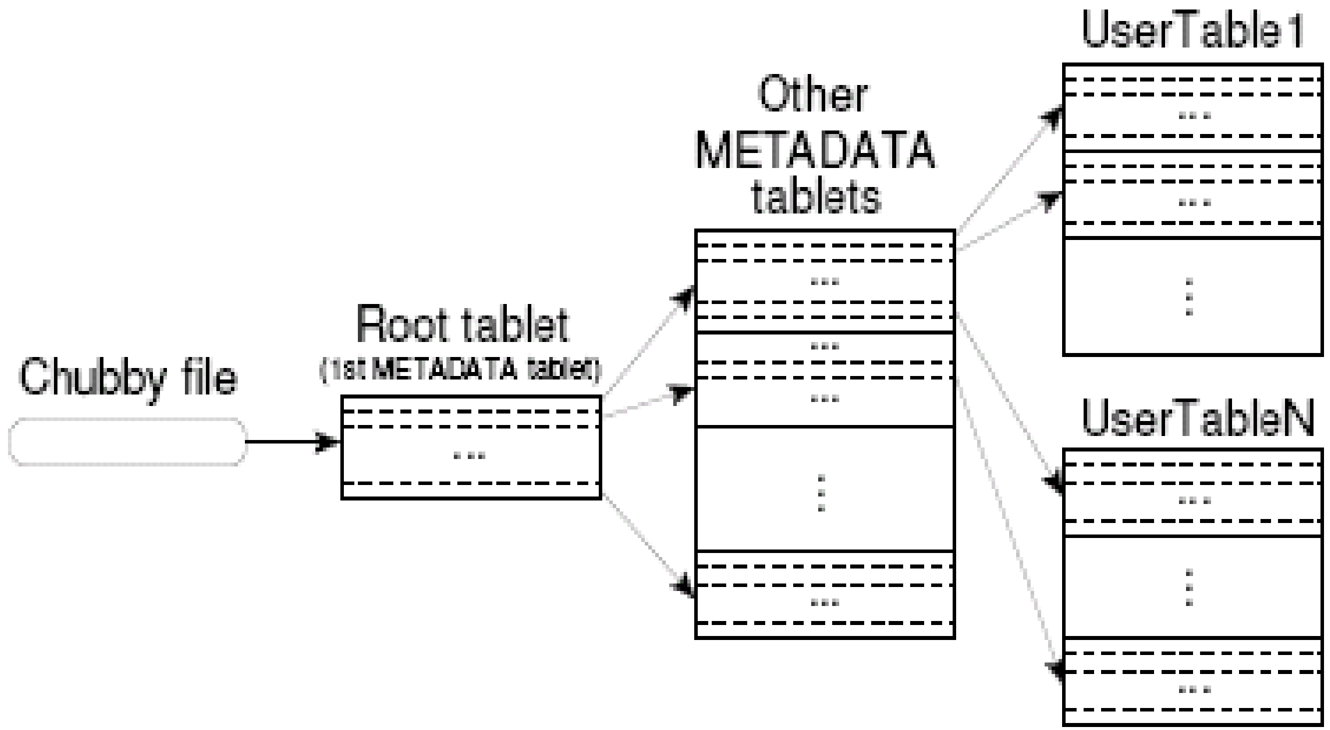
- BigTable uses a three-level hierarchy analogous to that of a B +- tree.
- The first level is a file stored in Chubby that contains the location of the root tablet.
- The root tablet contains the location of all tablets in a special METADATA table.

Locating a Table/Tablet/SSTable



- Each METADATA tablet contains the location of a set of user tablets.
- The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet’s table identifier and its end row.

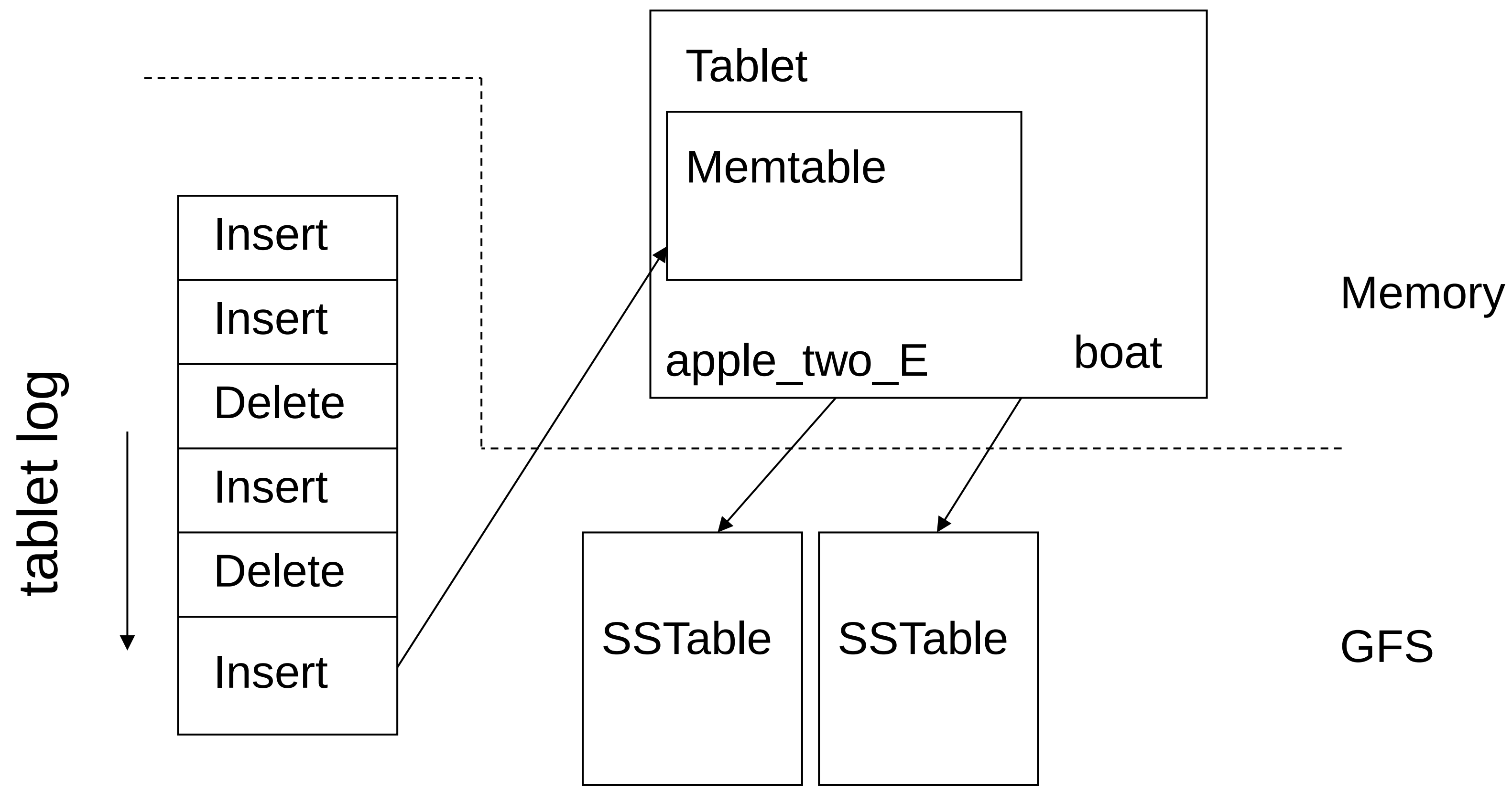
Locating a Table/Tablet/SSTable



- Each METADATA row stores approximately 1KB of data in memory.
- With a limit of 128 MB METADATA tablets, the three-level location scheme is sufficient to address 2^{34} tablets.

Editing a table

- Changes are logged, then applied to an in-memory memtable
  - May contain “deletion” entries to handle updates
  - Group commit on log: collect multiple updates before log flush



Compactions

- Minor compaction – convert the memtable into an SSTable
  - Reduce memory usage
  - Reduce log traffic on restart
- Merging compaction
  - Reduce number of SSTables
  - Good place to apply policy “keep only N versions”
- Major compaction
  - Merging compaction that results in only one SSTable
  - No deletion records, only live data

Immutability

- SSTables are immutable
  - simplifies caching, sharing across GFS etc
  - no need for concurrency control
  - SSTables of a tablet recorded in METADATA table
  - Garbage collection of SSTables done by master
  - On tablet split, split tables can start off quickly on shared SSTables, splitting them lazily
- Only memtable has reads and updates concurrent
  - copy on write rows, allow concurrent read/write

BigTable

- The data is always maintained consistently.
- Guarantee coming from using Chubby
- The model is naturally partitioned horizontally.
- But, availability may be in question.
- There could be brief downtimes, due to
  - Chubby not being available
  - network problems

Example Applications Using BigTable

- Goole Analytics:
  - Track various metrics such as traffic patterns, site-tracking reports, …
- The raw click table (˜200 TB) maintains a row for each end-user session.
- The row name is a tuple containing the website's name and the time at which the session was created.

Example Applications Using BigTable

- Goole Personalized Search:
- Records user queries and clicks across a variety of Google properties such as web search, images, and news.
- Users can browse their search histories to revisit their old queries and clicks, and they can ask for personalized search results based on their historical Google usage patterns.
- Personalized Search stores each user's data in Bigtable.
- Each user has a unique userid and is assigned a row named by that userid. All user actions are stored in a table.