Consistent Join Queries for Cloud Data Stores

Zhou Wei [1,2]

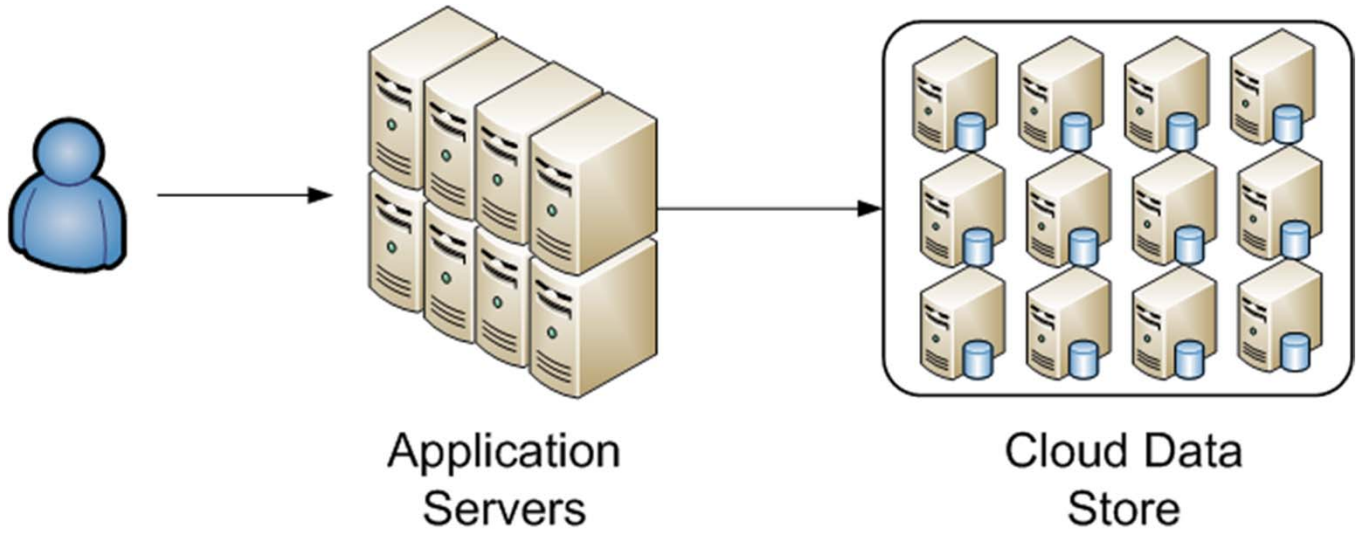
Guillaume Pierre[1], Chi-Hung Chi[2]

[1] VU University Amsterdam

[2] Tsinghua University Beijing

Background

- Cloud Computing Platforms for Web Apps
  - Unlimited resources
  - Scalable NoSQL data stores
  - Pay-as-you-go



Problems of Cloud data stores

- Problem 1: No Join queries
  - Only support queries within a single table
  - No queries across tables
- Problem 2: Relaxed data consistency
  - Single-row transaction only
  - No consistency guarantee across multi-items
- Web apps need consistent join queries

Why Web App needs join?

- Data Normalization → Foreign-key Equi-joins

Post

PK	Title	Comment
1	IPhone	Cool
2	IPhone	Nice
3	IPhone	Expensive

Post

PK	Title
1	IPhone

Comment

PK	Post_id	Comment
1	1	Cool
2	1	Nice
3	1	Expensive

Why consistent?

- Application correctness is not an option
- Hard to maintain correctness with an inconsistent data store
  - Require skillful programmers
  - Error-Prone
  - Cost more time

Alternatives

- **SQL** databases
  - Centralized
  - Full Replication
  - **NOT** scalable to update workload
  - Complicated queries
  - ACID Transaction

MySQL
Oracle
…

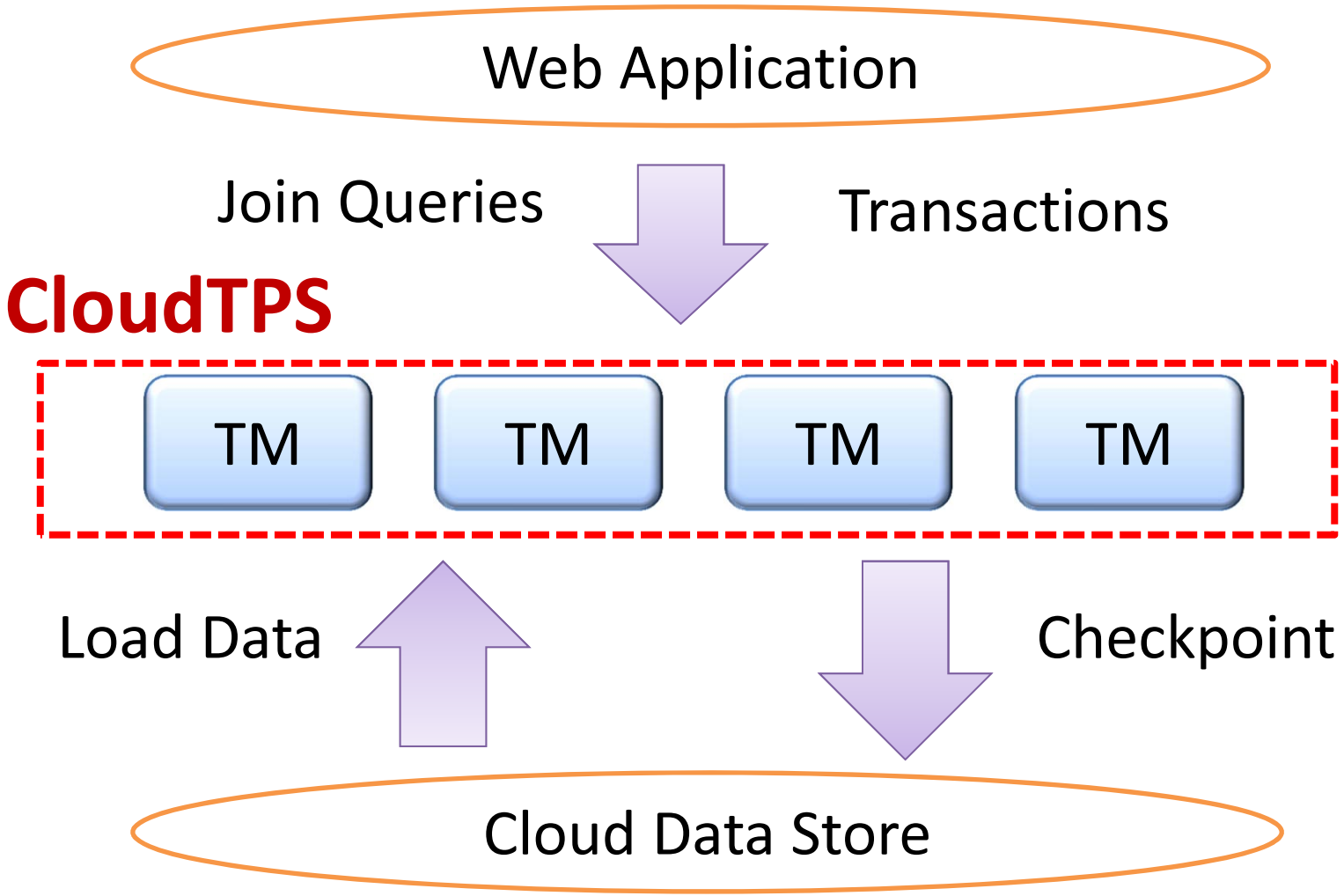
- **NoSQL** data stores
  - Distributed
  - Partitioned
  - Scalable to update workload
  - **No Join Queries**
  - **Single-row Transaction**

Google Bigtable
Amazon SimpleDB
Cassandra
…

Our Position

- Scalable NoSQL data stores supporting equi-joins queries for Web applications
- Join query as a ACID transaction
- Queries of Web applications access only a small portion of data
  - Also true for join queries
  - All of them are foreign-key equi-joins

How it works?


Web Application
Join Queries
Transactions
CloudTPS
TM
TM
TM
TM
Load Data
Checkpoint
Cloud Data Store

Outline

- Join Algorithm
- Transaction Protocol
- Performance Evaluation
- Conclusion

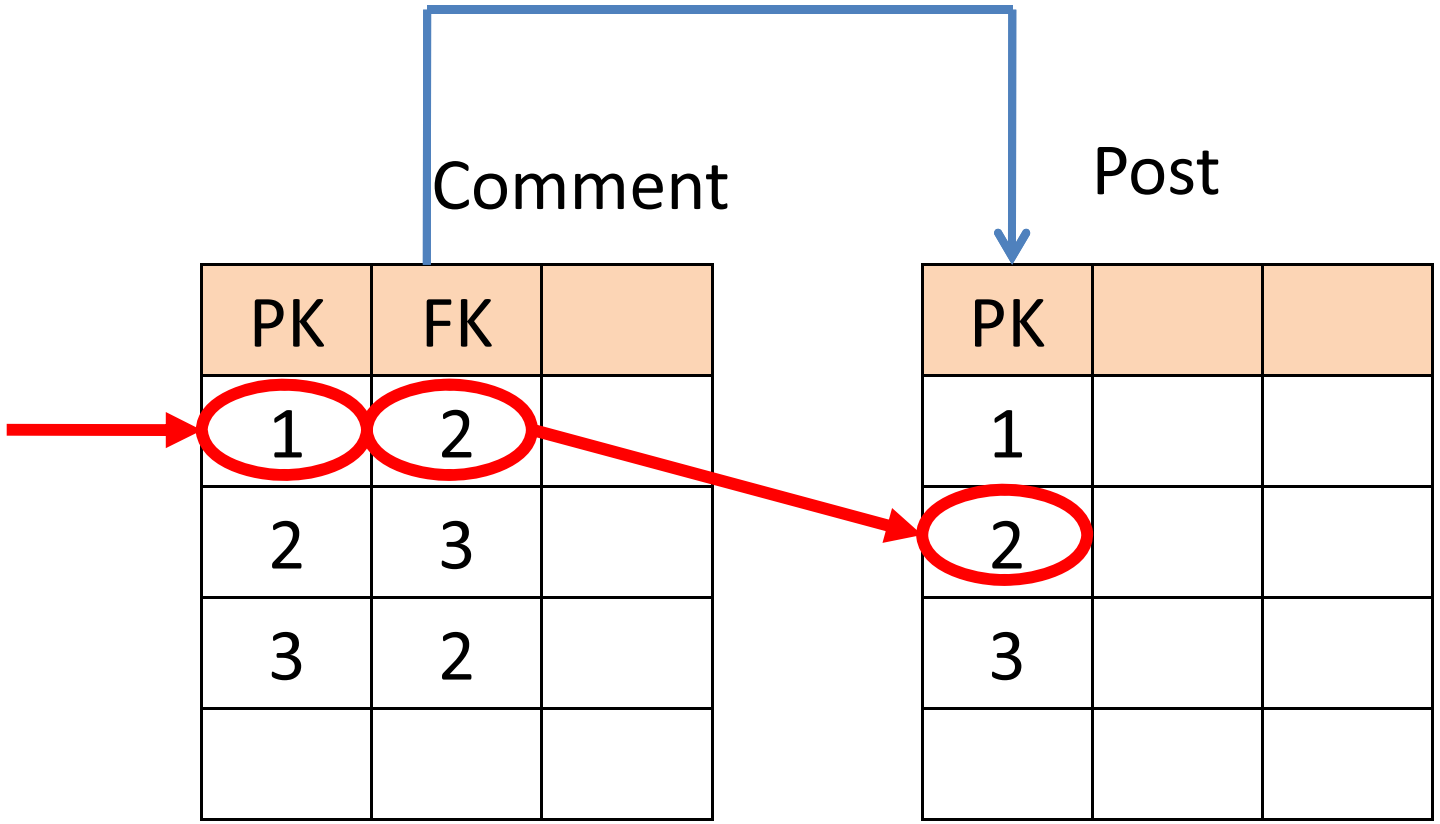
Join Algorithm

- Web applications are response-time sensitive
  - Table scan is not an option
- The basic idea
  - Linking records by FK relationships in advance
  - Each join query has well identified “root records”
  - Begin from accessing each “root record”
  - Recursively obtaining its related records

Link records for forward join

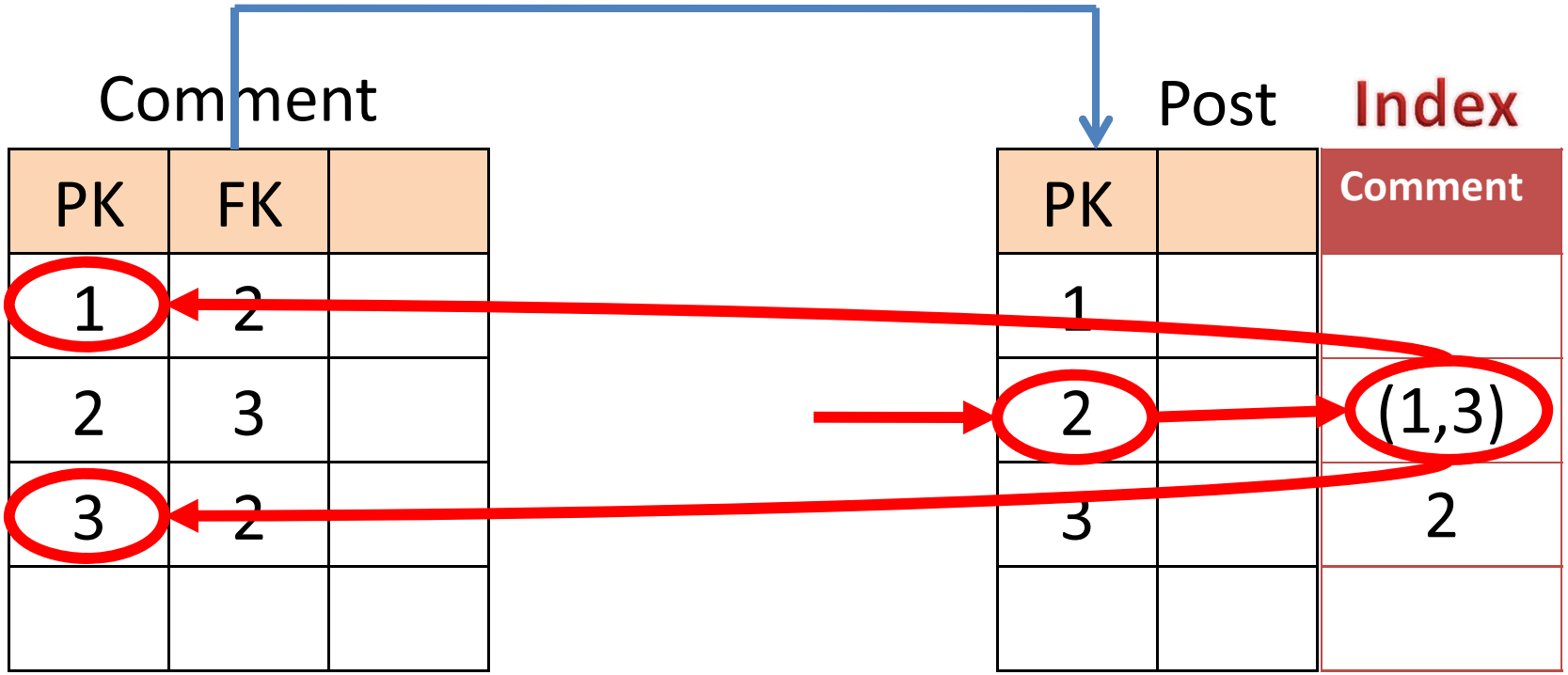
Given referring row → identify referenced row

(FK) (PK)



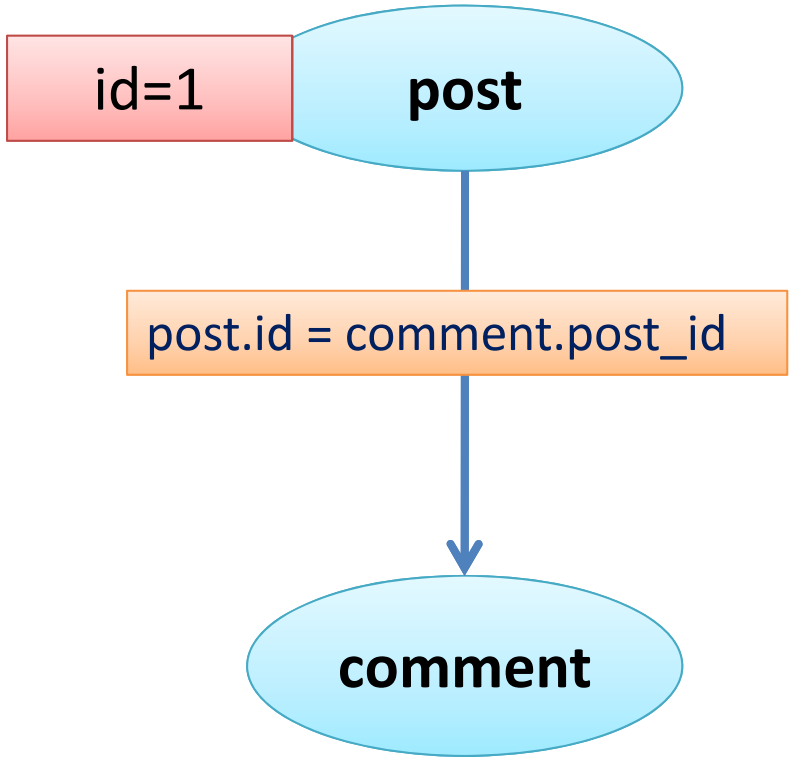
Link records for backward join

Give referenced row → identify referring rows
(PK) (FK)



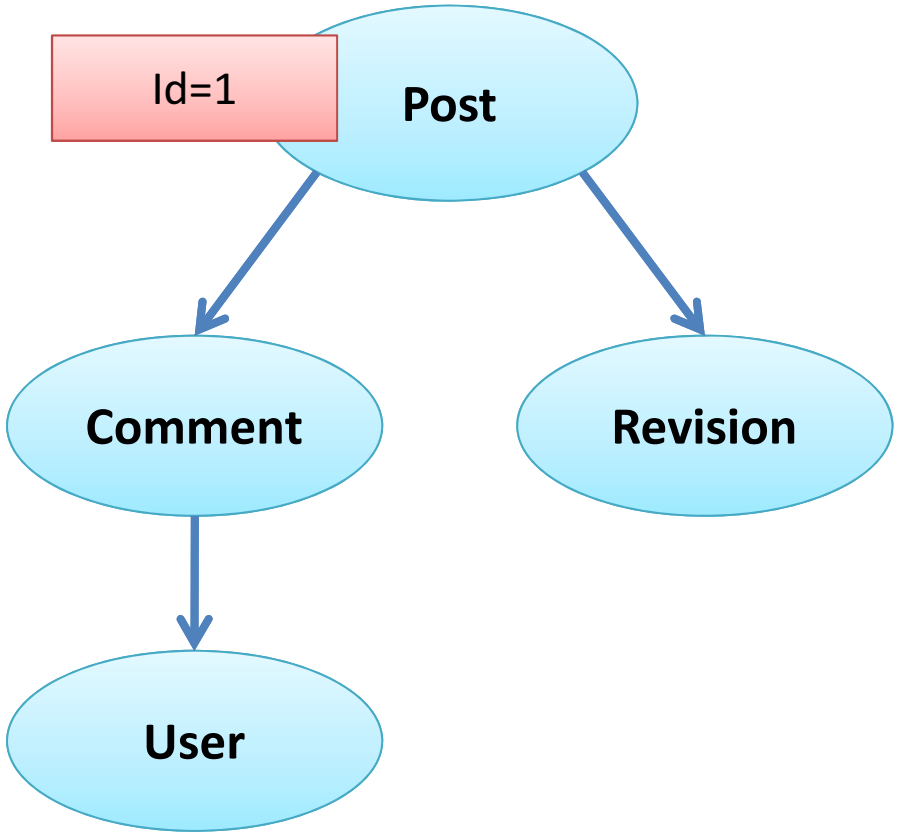
Joining two tables

```
SELECT * FROM post,
comment WHERE
post.id = 1 and
comment.post_id =
post.id
```



Multi-Joins

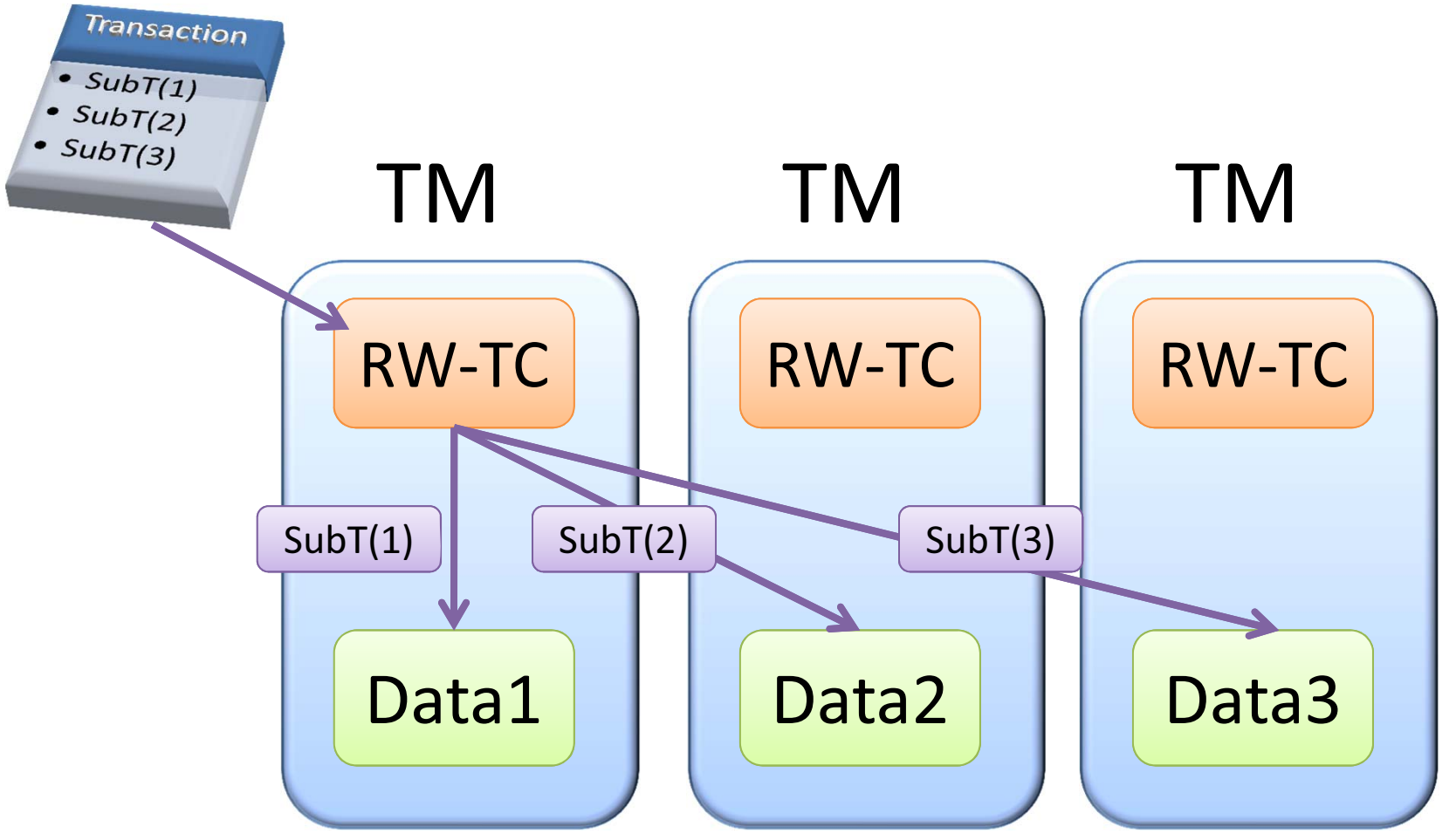
- The table of the "root record" is the root
- Recursively join child tables
- Length of critical execution path = 2



Outline

- Join Algorithm
- Transaction Protocol
- Performance Evaluation
- Conclusion

Distributed Transaction

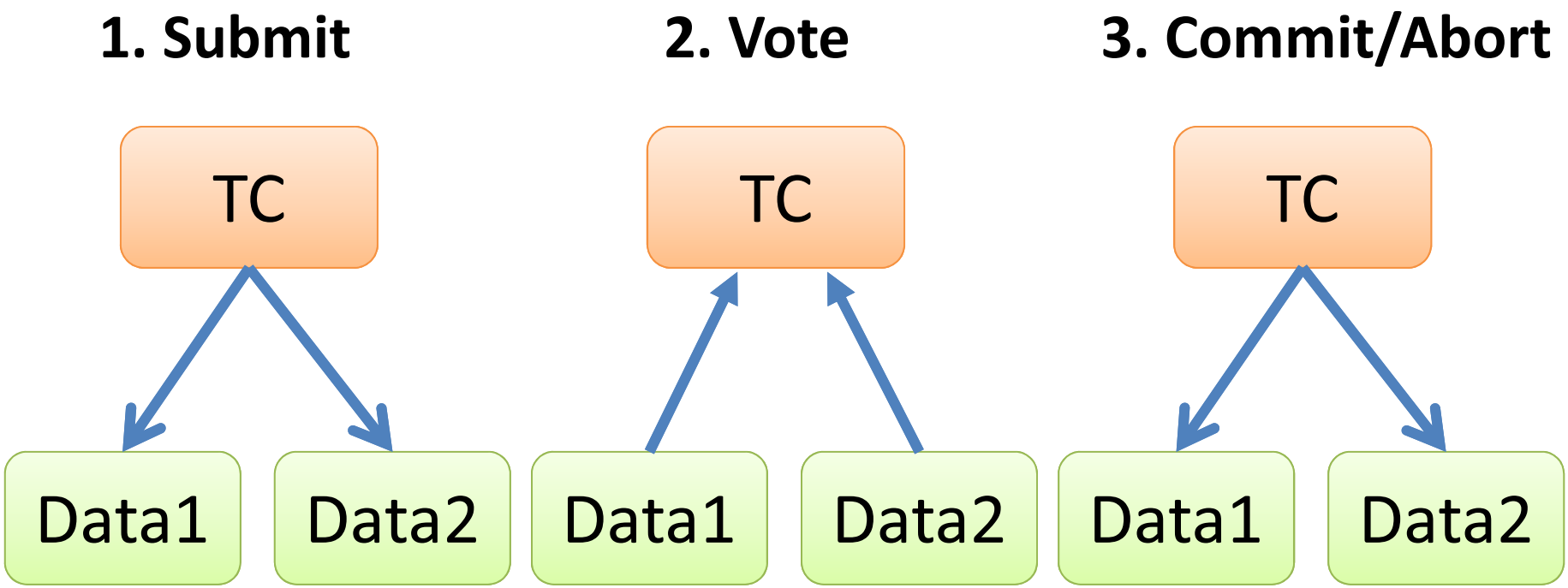


2 Phase Commit (2PC)

Problem

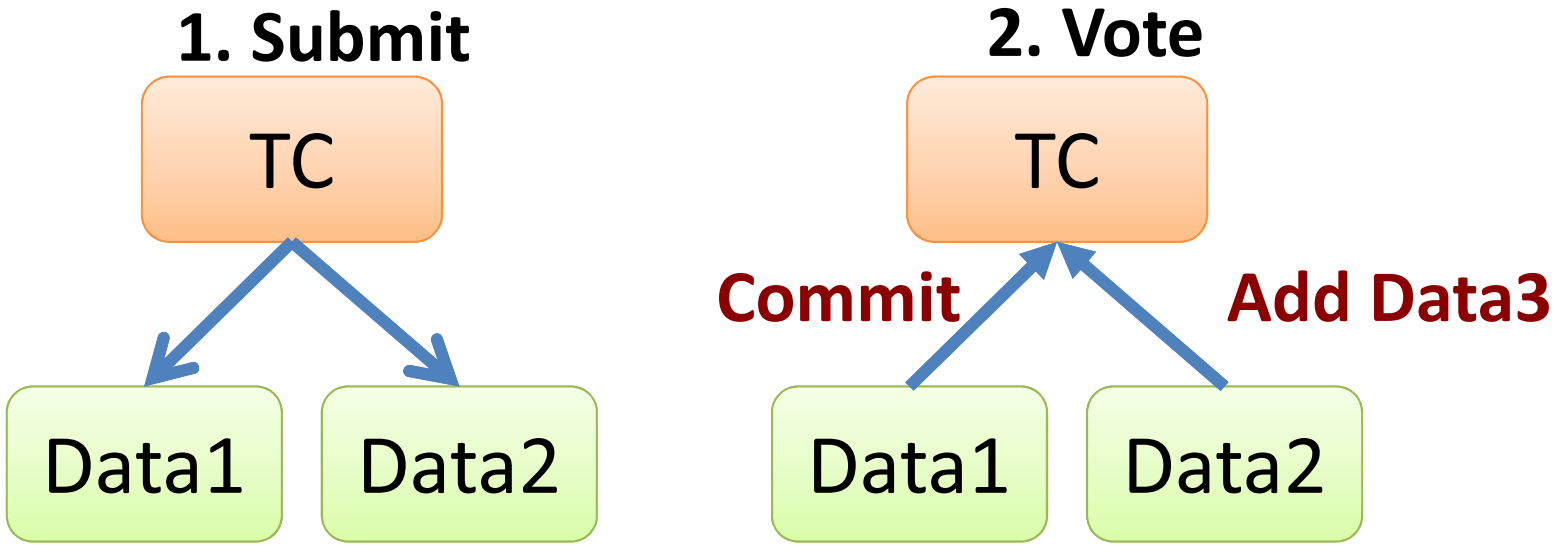
- 2PC requires all records identified in advance
- Join queries/index management identify accessed records on the fly
- Solution: extend 2PC to allow adding records dynamically

The Original 2PC

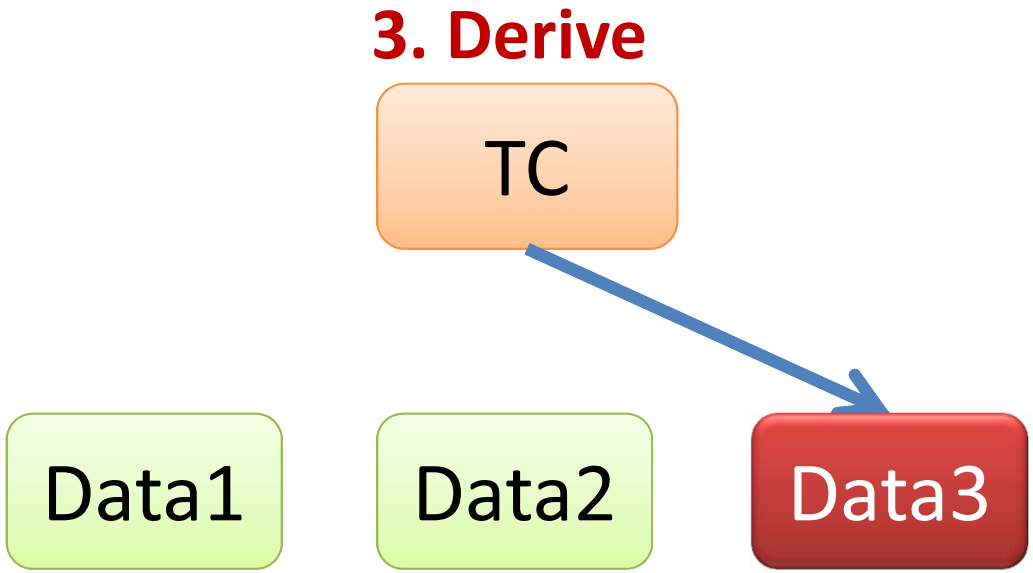


Extended 2PC

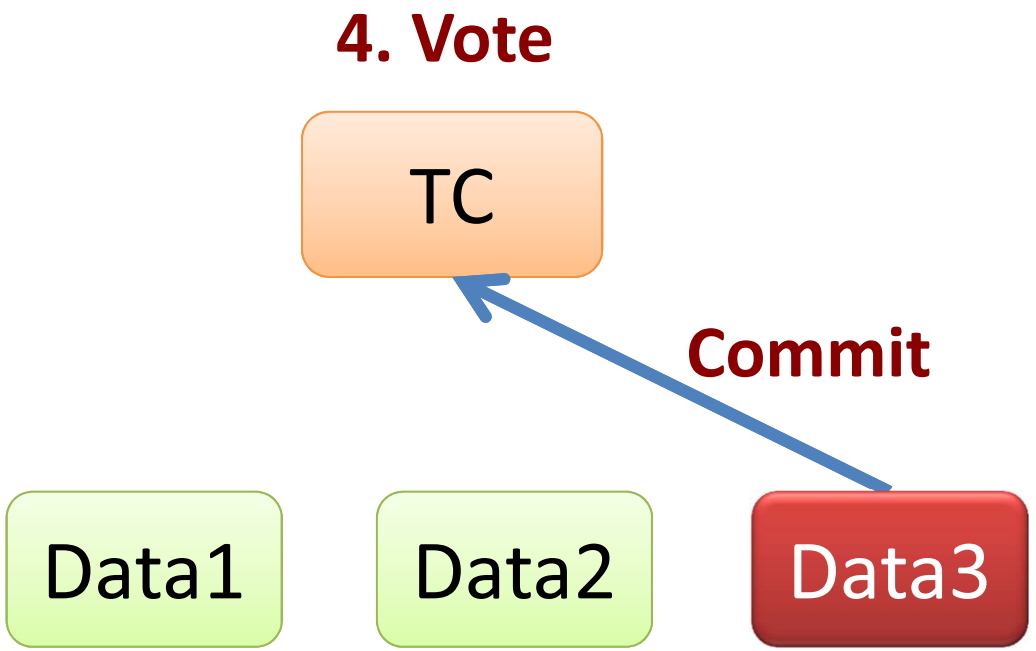
- Dynamically add data items to the transaction



Extended 2PC

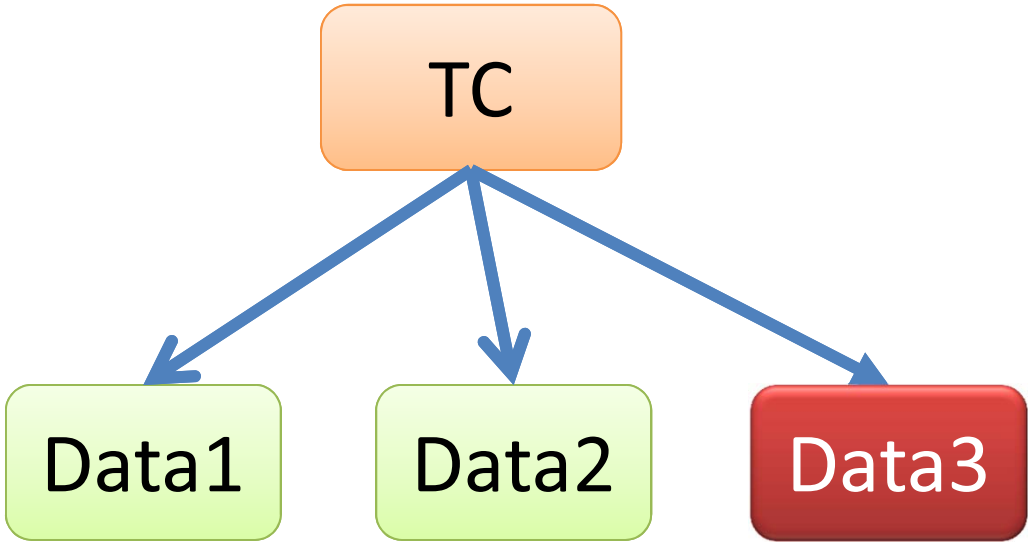


Extended 2PC



Extended 2PC

5. Commit



Outline

- Join Algorithm
- Transaction Protocol
- Performance Evaluation
- Conclusion

Experiment Setup

- Implement a prototype of CloudTPS
  - Java Servlet
  - Deployed in Tomcat v6.0
- Environments
  - 1. Das3+Hbase+Tomcat (99% reqs < 100ms)
  - 2. EC2+SimpleDB+Tomcat (90% reqs < 100ms)
    - High-CPU medium instances

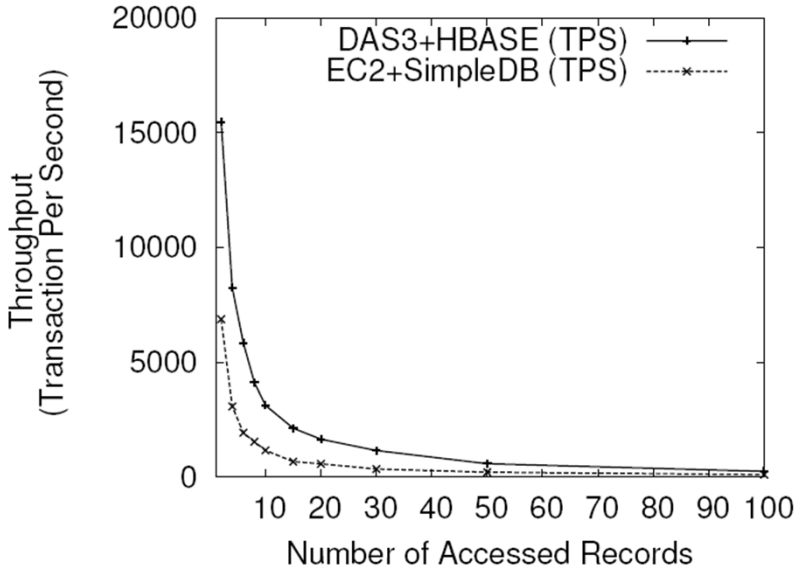
Micro-benchmarks

- Generate specific workload
  - Containing only Join queries/RW transactions
  - Changing the number of accessed data items
  - Changing the length of critical execution path
  - Switch between Update indexes/Not update
- All configured with 10 CloudTPS nodes
- Measuring maximum sustainable throughput
  - Under response time constraint

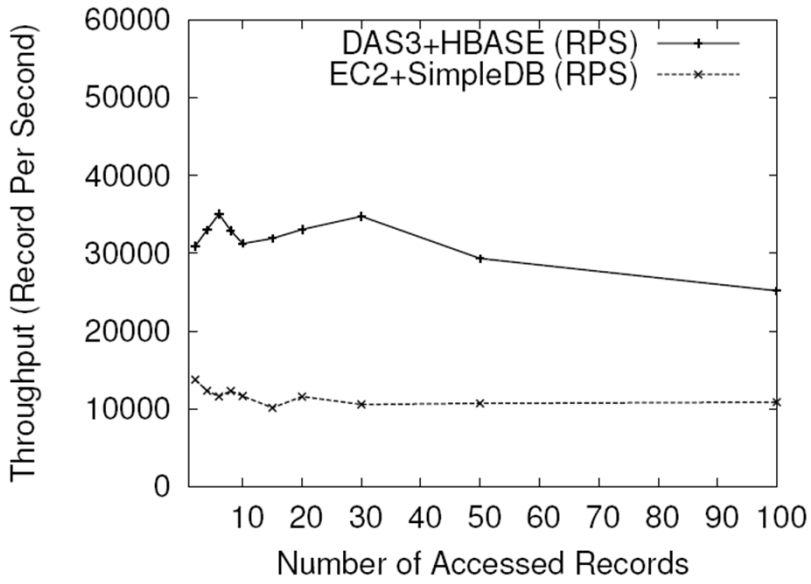
Two-table join evaluation

Increasing number of accessed records in a transaction

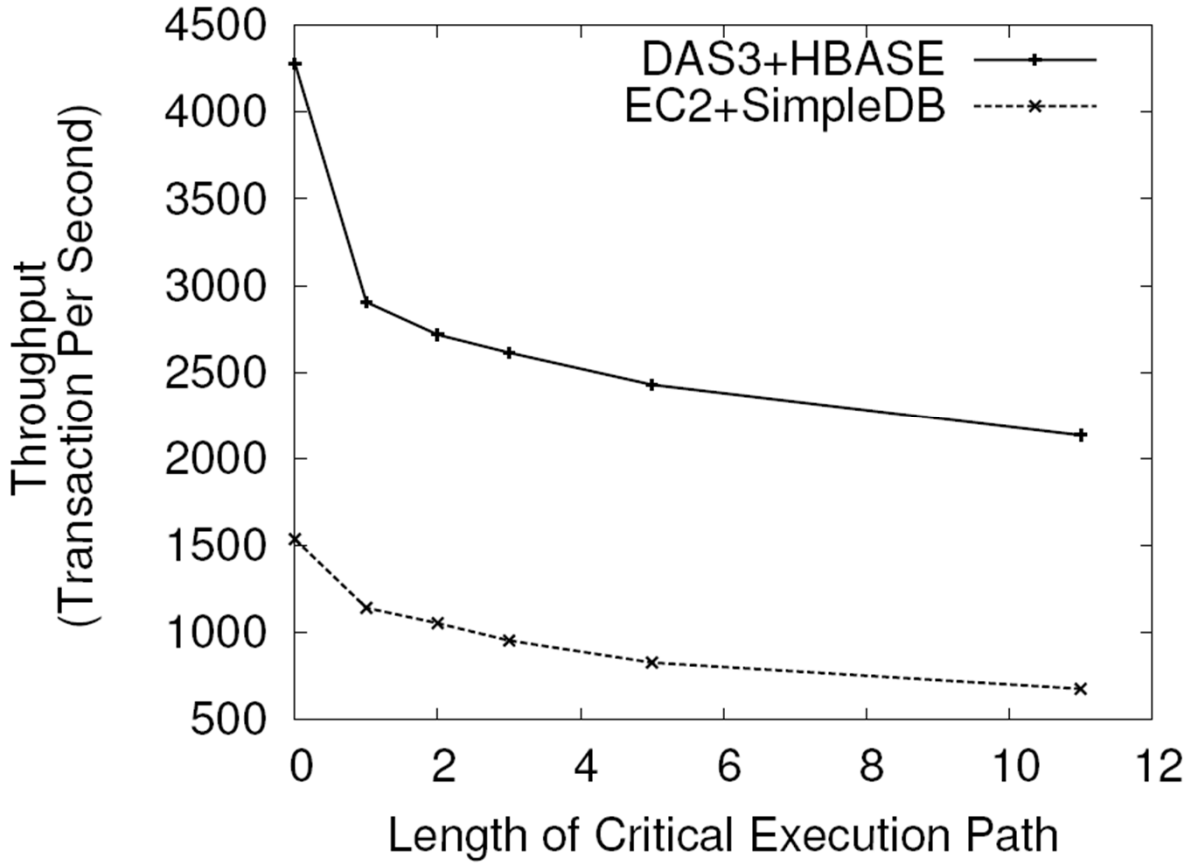
Transaction Per Second



Record Per Second



Multi-Joins Evaluation

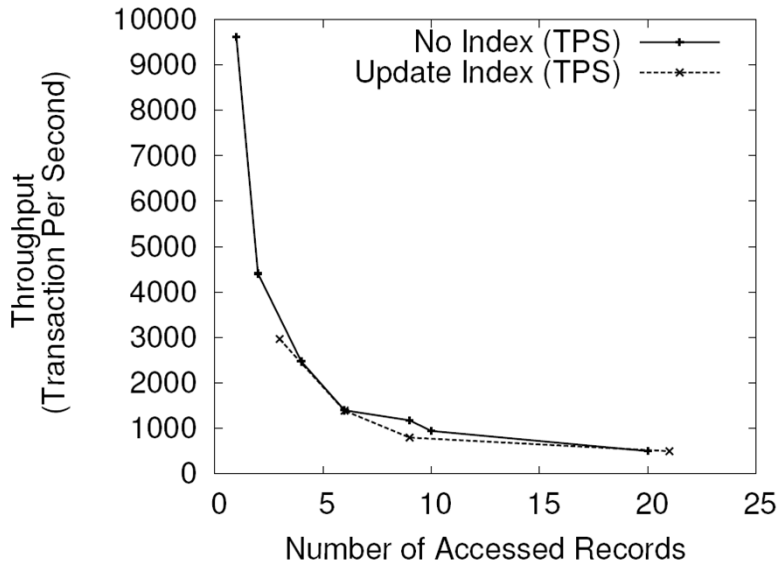


Increasing the height of join-query tree

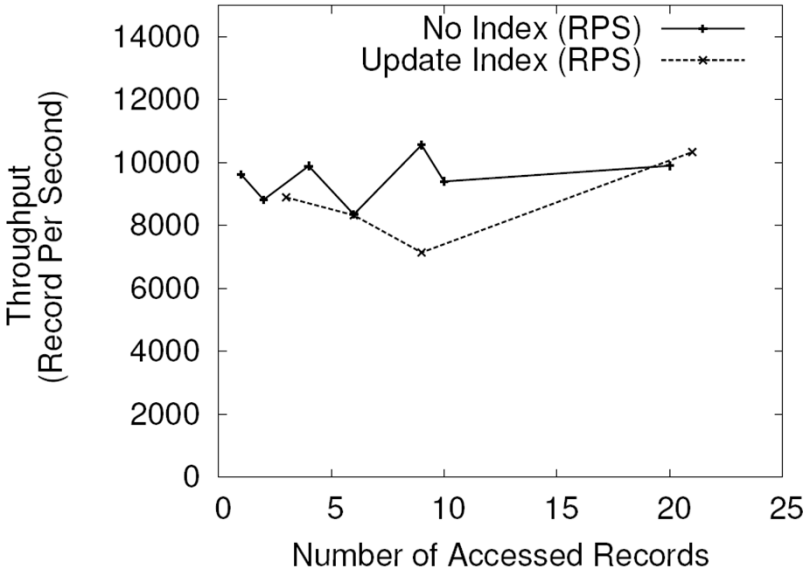
All join queries access 12 records

RW transactions evaluation

Transaction Per Second



Record Per Second

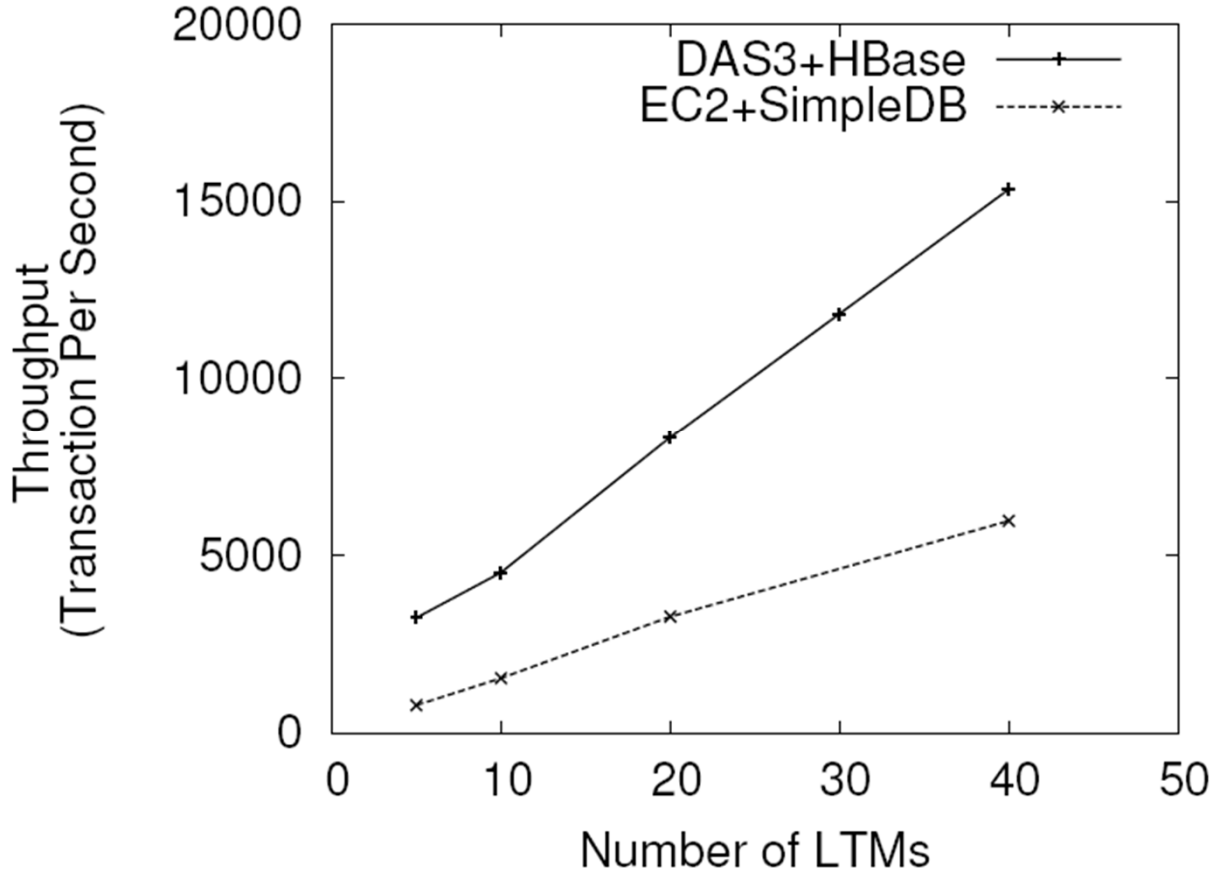


Increasing number of accessed records in a transaction

Scalability Evaluation

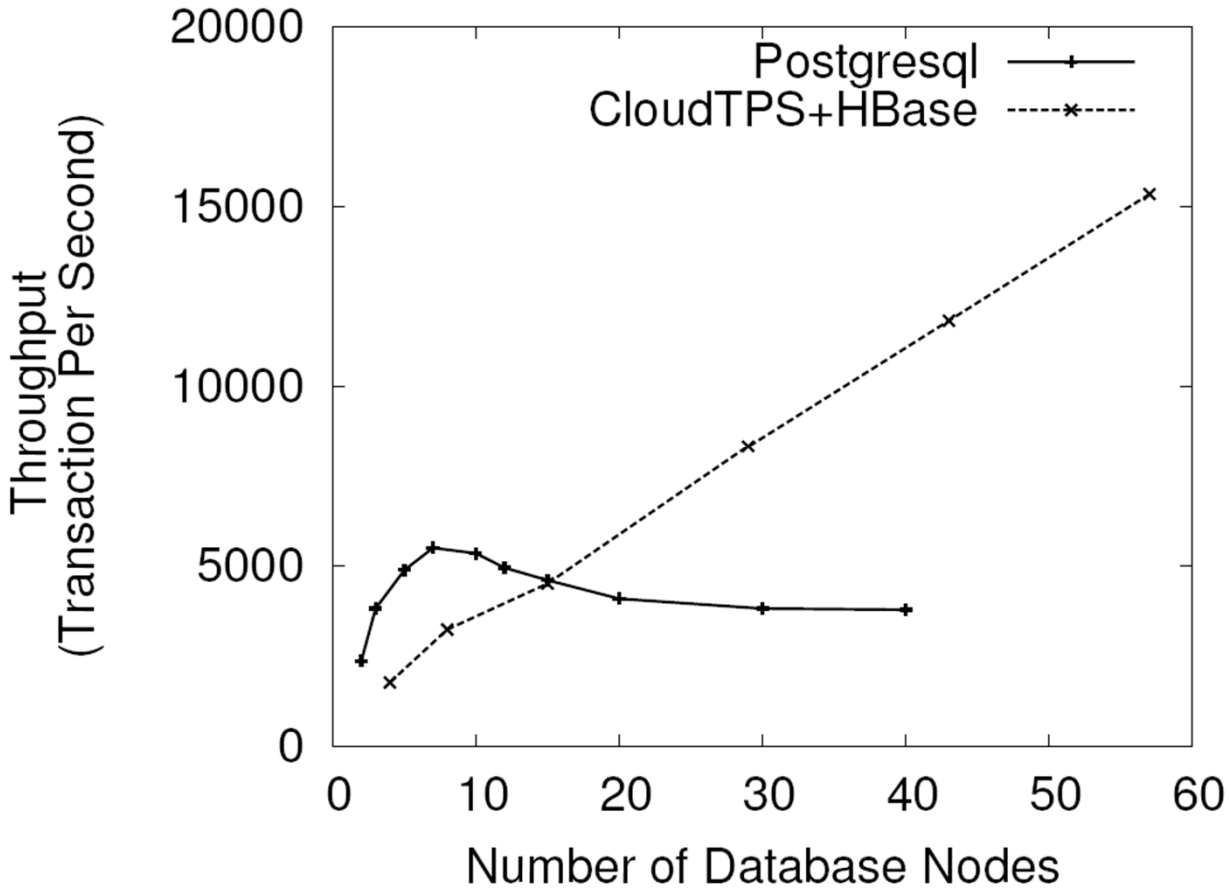
- Macro-Benchmark
  - Derived from TPC-W, simulating an online book store
  - Our workload includes only the join queries and RW transactions
  - No simple queries

Scalability Results



Linearly Scalability: Any increase of workload can be addressed by adding more machines

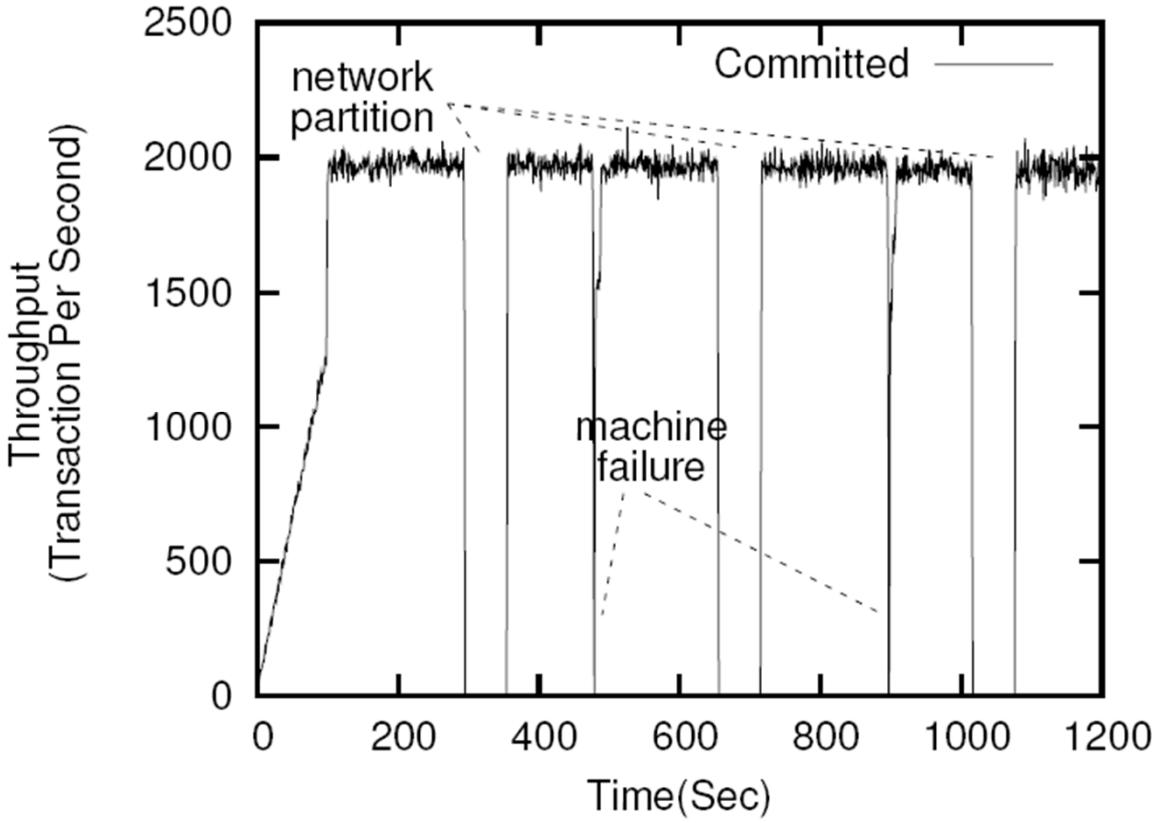
CloudTPS vs. Postgresql



Postgresql: binary-replication

Fault Tolerance

- Recovers from 3 network partitions and 2 machine failures



Conclusion

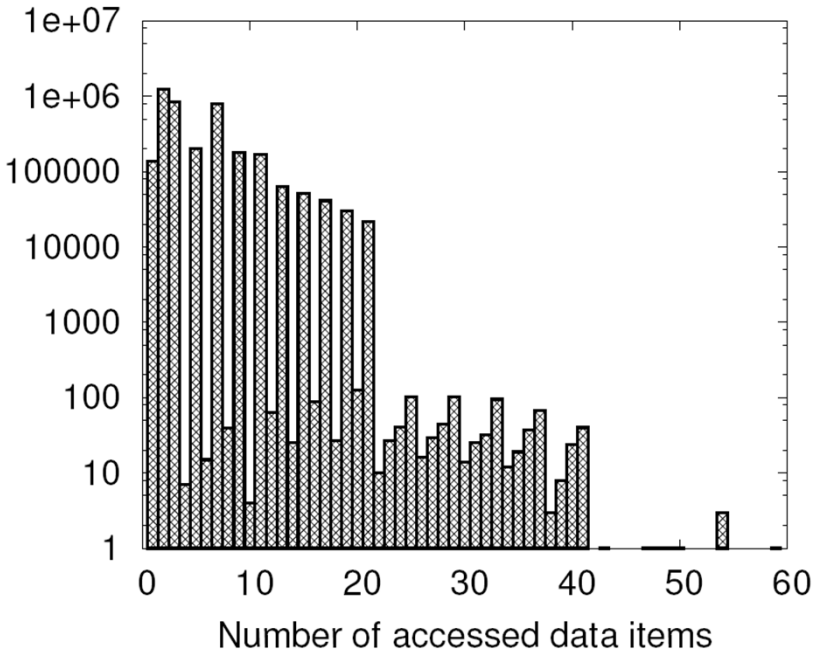
- We demonstrate supporting consistent join queries over partitioned and distributed data
- Join queries from Web applications only access a small portion of data
- It scales linearly even under an intensive workload of only join queries and read-write transactions

Thank you! ☺

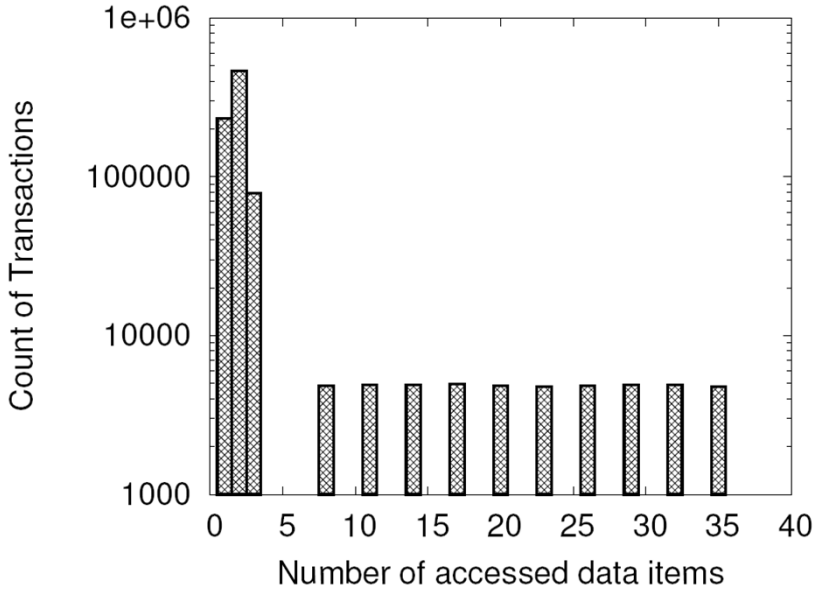
Questions?

Details in Macro-benchmark workload

Join queries

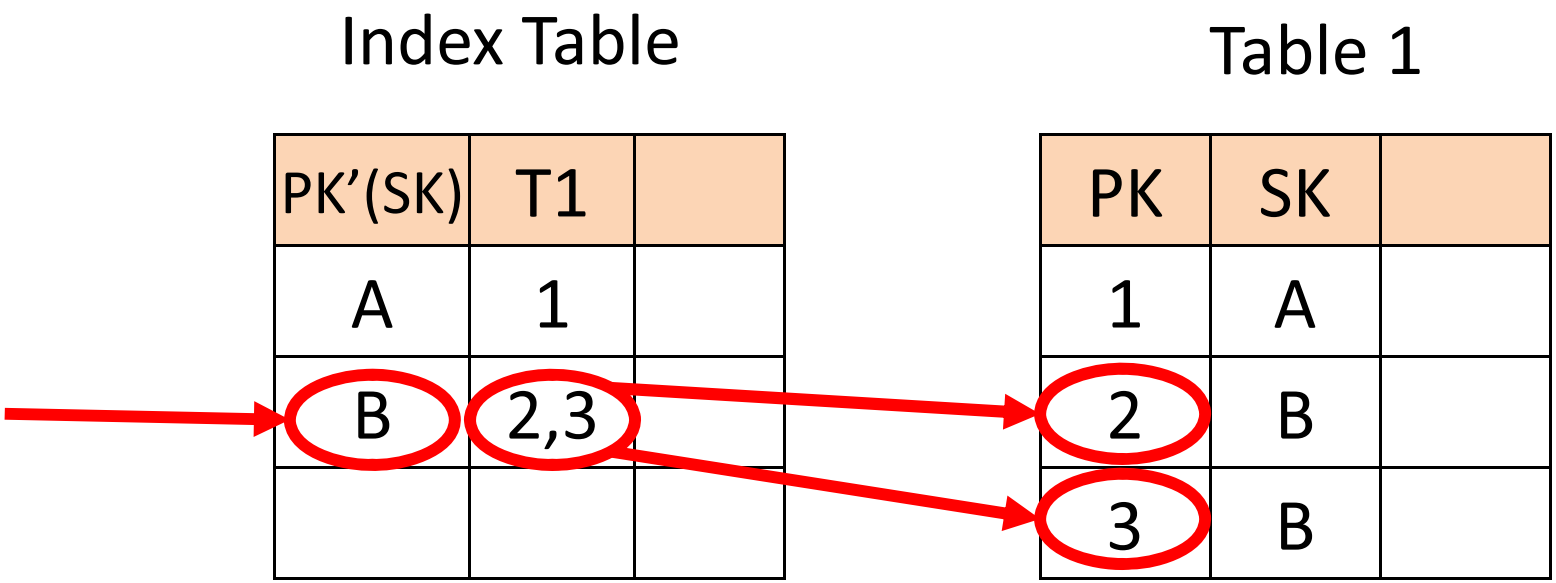


RW transactions



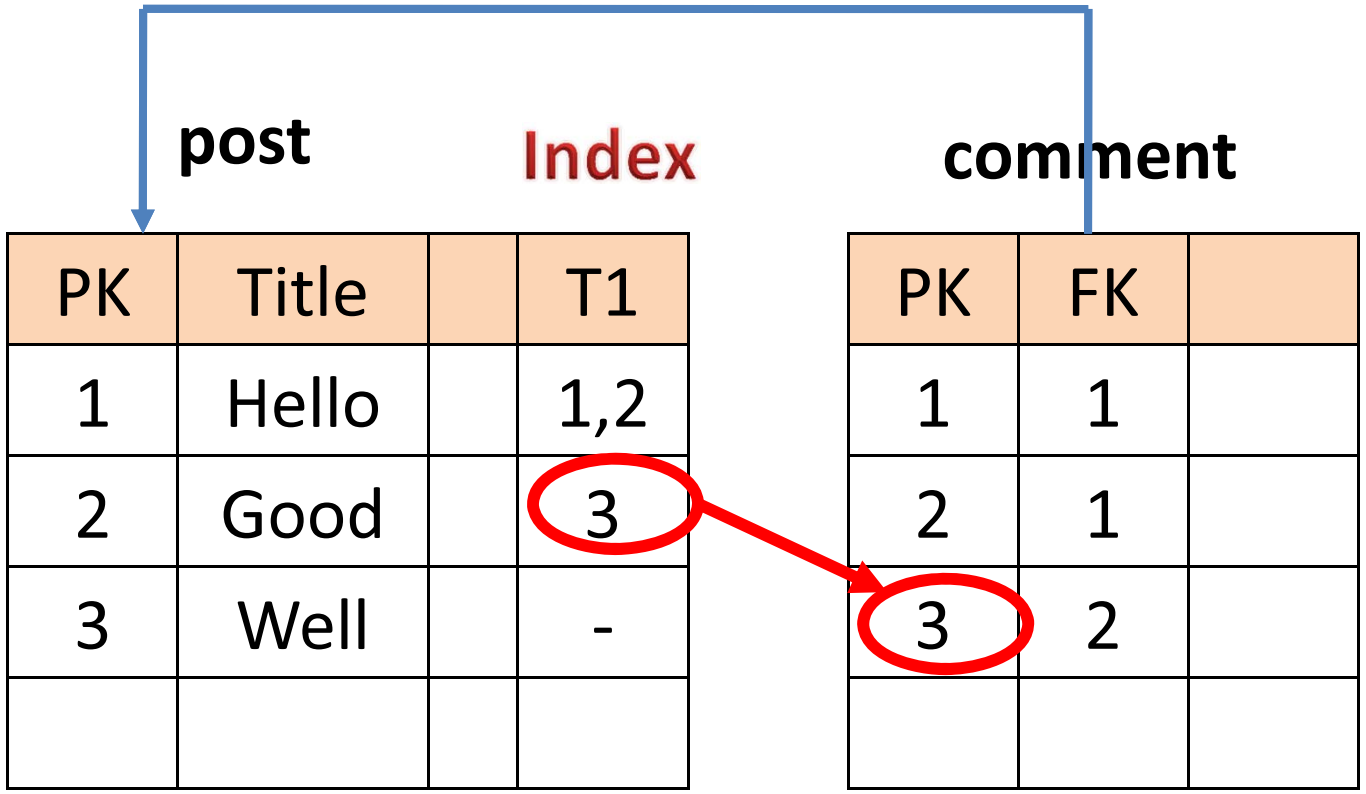
Secondary-key Queries

- Select * from Table1 Where SK = ‘B’
- Create a separate index table for the SK
- Translate into a join query



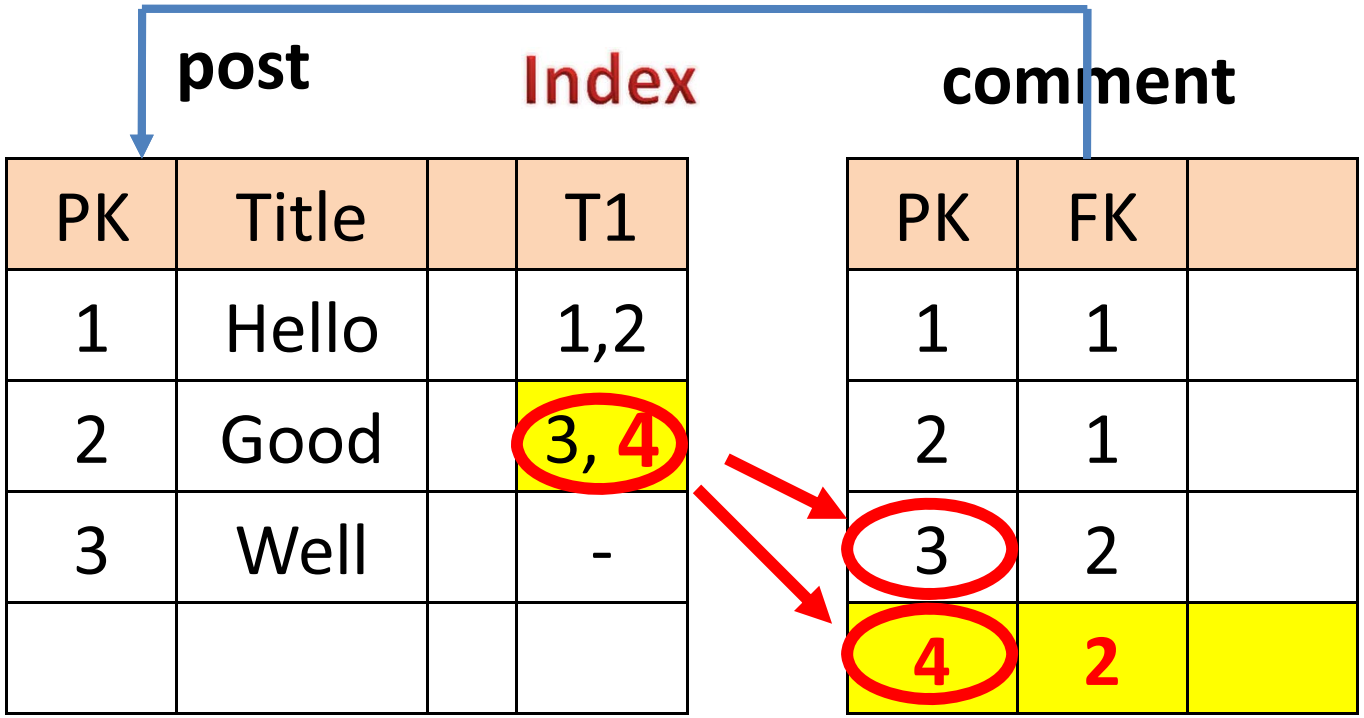
Index Management

- Indexes link the related records



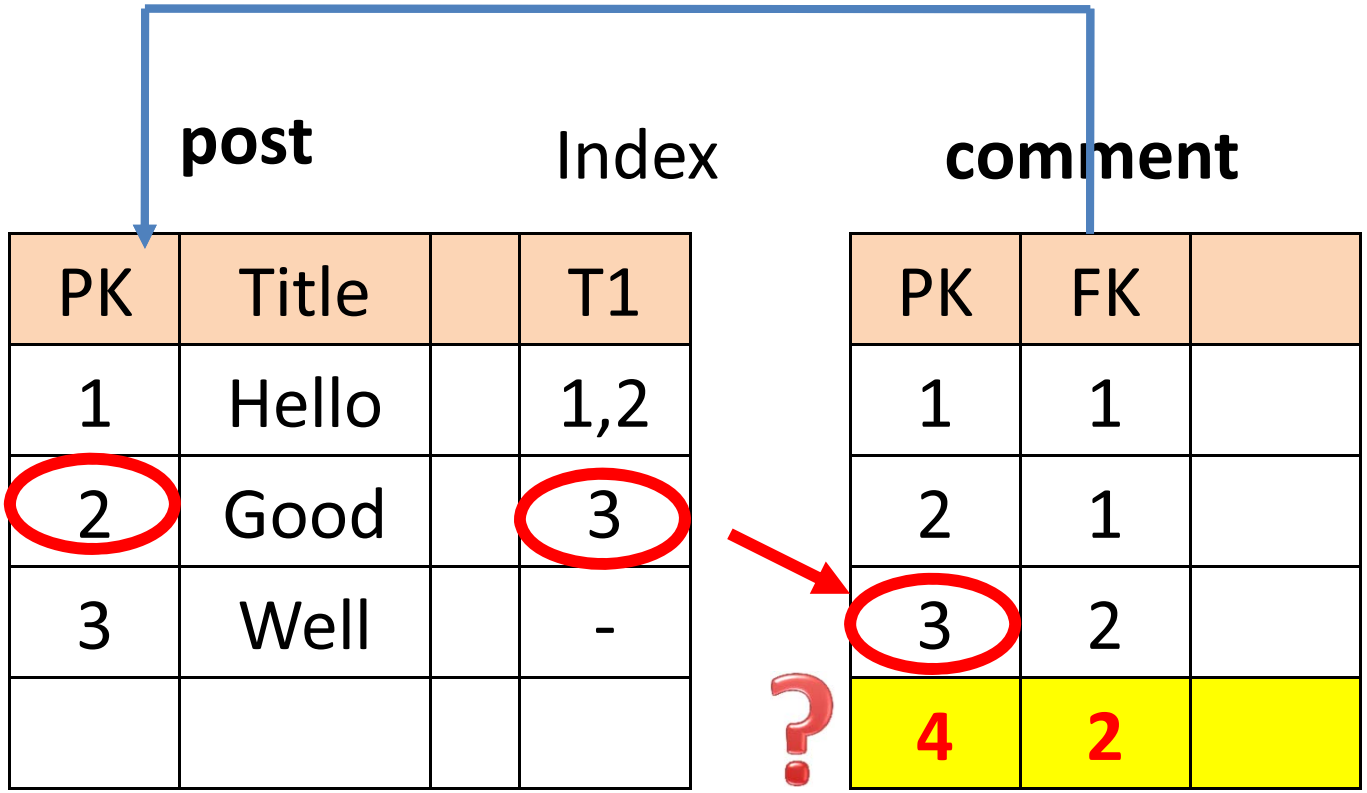
Index Management (cont.)

- Case: a new comment inserted
- Updating indexes atomically



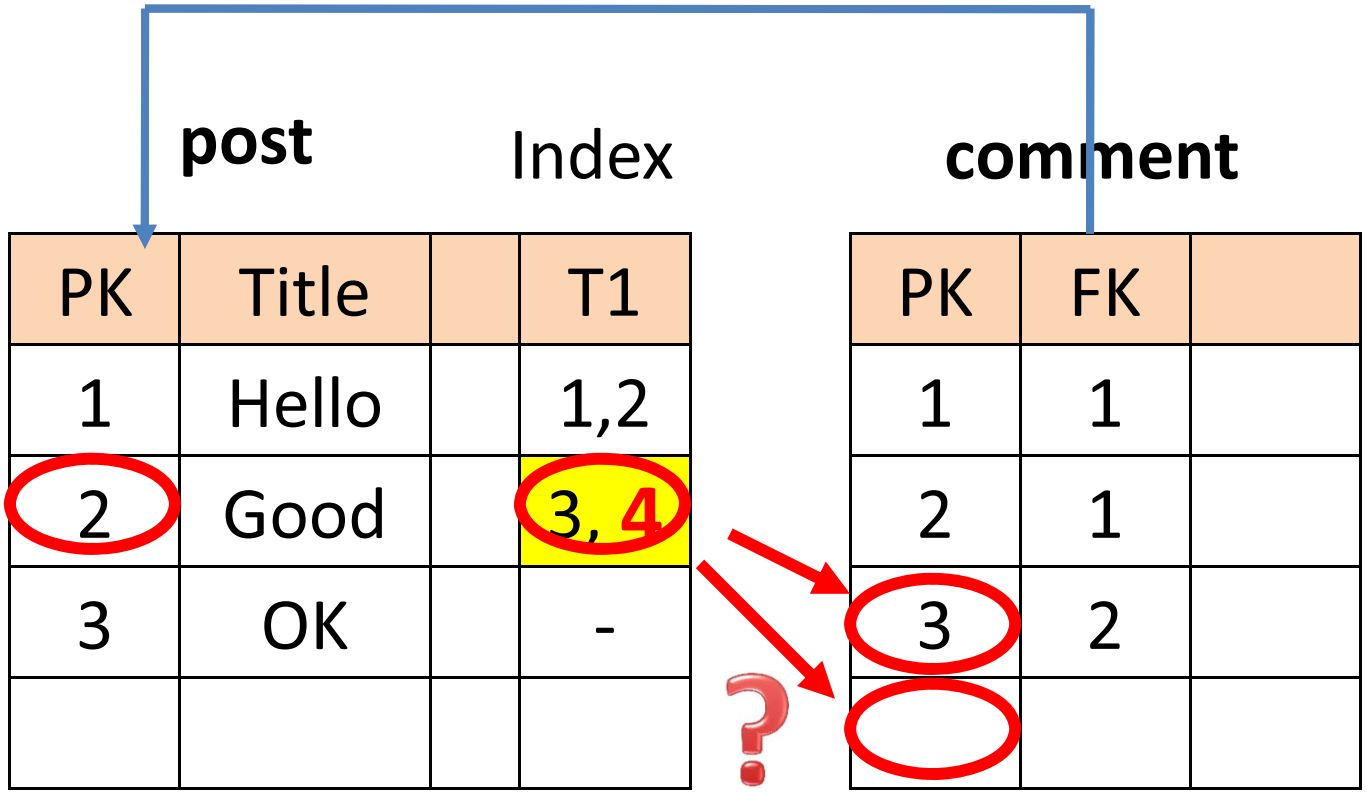
Index Management (cont.)

- Consistency Violated Case 1



Index Management (cont.)

- Consistency Violated Case 2



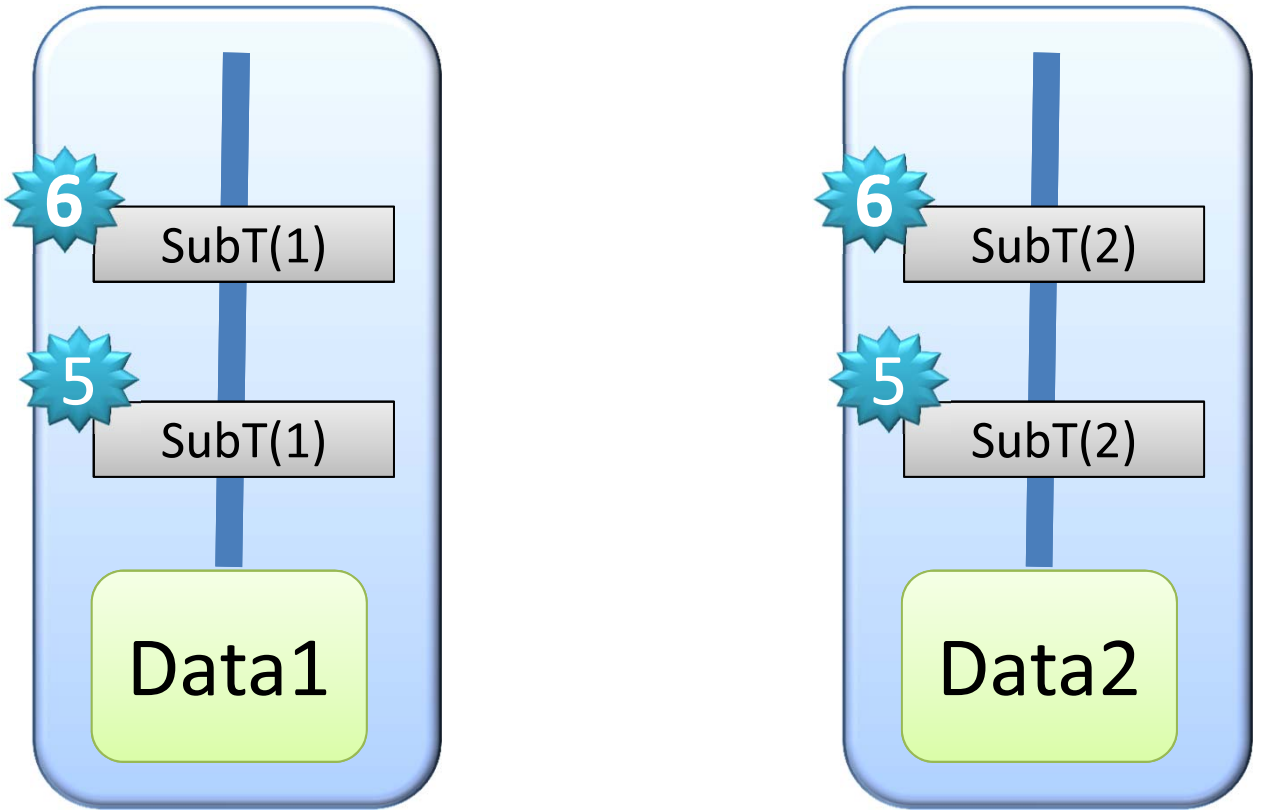
Fault tolerance

- Maintain ACID during server failure
  - Transactions that reach an agreement of “COMMIT” should Commit
  - Abort other transactions
- Replication to N transaction managers
  - Values of Data Items
  - States of Transactions (Vote result and “redo logs”)
- Failover of a failed server
  - Transfer ownership of data items
  - Transfer leadership of transactions being coordinated

Optimization for Join Queries

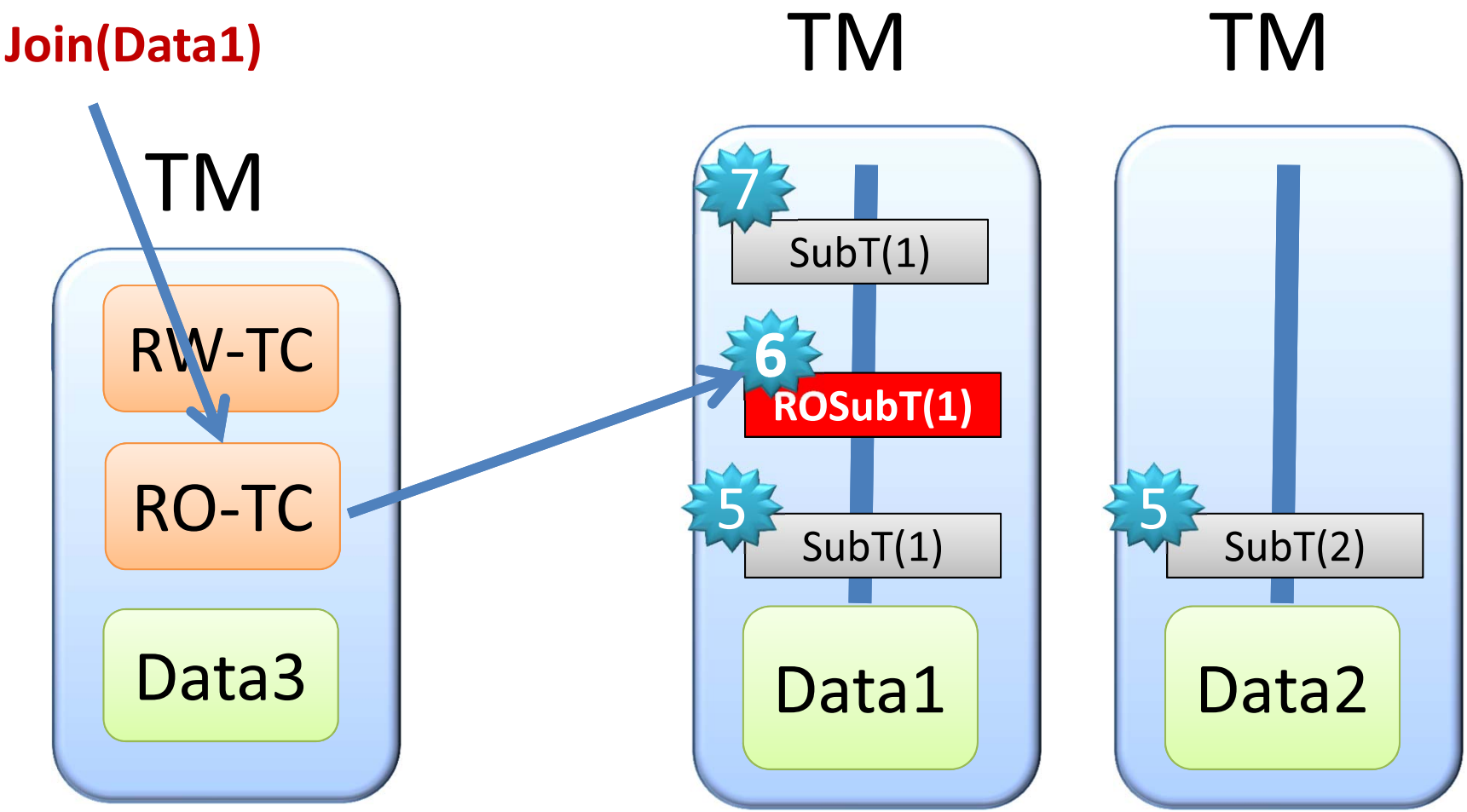
- Join Query is a Read-only transaction
  - Remove the final Commit phase
  - Sub-transactions removed immediately after local execution
  - RO transactions will not block other transactions

Concurrency Control

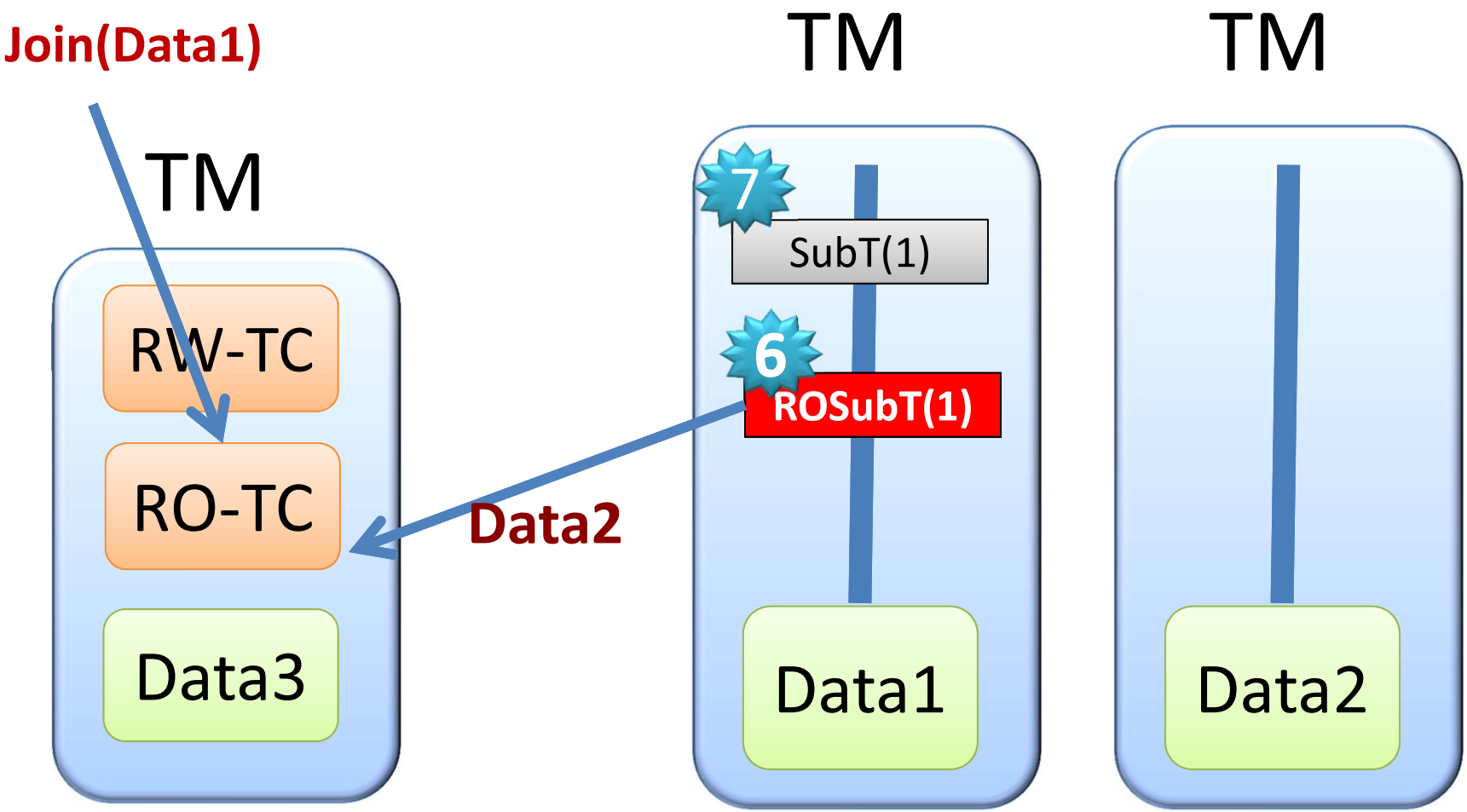


Timestamp Ordering

Read-Only Transaction



Read-Only Transaction (Cont.)



Read-only Transaction (cont.)