Coordination Avoidance in Distributed Databases

Peter Bailis (Stanford University)

Chapter 4
Coordination Avoidance and Weak Isolation

Seminar №2
Gabeydulin Ramis,
14.02.2018

Coordination

- Databases are traditionally tasked with maintaining, informally, “correct” data—that is, data that obey some semantic guarantees about their integrity.
- Thus, during concurrent access to data, a database ensuring data correctness must therefore decide which user operations can execute simultaneously and which, if any, cannot. Informally, we say that two operations within a database must *coordinate* if they cannot execute concurrently on independent copies of the database state
- The classic answer to maintaining application-level invariants is to use **serializable isolation**: execute each user’s ordered sequence of operations, or transactions, such that the end result is equivalent to some sequential execution
- The problem is that serializable semantics require **coordination**

Invariant confluence

- **The goal** of the work is to study the design of database systems that provide coordination-free execution, which implies availability, low latency and scalability.
- To achieve this goal, the author proposed a new, general rule for determining whether a coordination-free implementation of a given safety property exists, called *invariant confluence*
- **Invariant confluence** determines whether the result of executing operations on independent copies of data can be combined (or "merged") into a single copy of database state. Given a set of operations, a safety property that we wish to maintain over all copies of database state, and a merge function, invariant confluence tells us whether coordination-free execution is possible.

Weak isolation in modern databases

- In the early 1970s, database systems have offered a range of weak isolation models
- None of these weak isolation models guarantees serializability, but their benefits to concurrency are frequently considered by database administrators and application developers to outweigh costs of possible consistency anomalies that might arise from their use
- To better understand the prevalence of weak isolation, author surveyed the default and maximum isolation guarantees provided by 18 modern databases
- Only three out of 18 databases provided serializability by default, and eight did not provide serializability as an option at all.

Database	Default	Maximum
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S

RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read

Invariant confluence analysis for weak isolation guarantees

- These non-serializable transaction semantics are widespread use in today's database engines and therefore provide an attractive target for invariant confluence analysis
- The author determined the invariant confluence of a number of weak isolation guarantees providing an informal explanation and example for each guarantee
- For each invariant confluent guarantee, he also offered proof-of-concept coordination-free algorithms
- These algorithms are not necessarily optimal or even efficient: the goal is to illustrate the existence of algorithms.

System model and notation

- In our examples, we exclusively consider read and write operations, denoting a write of version v_i with unique timestamp i to data item d as $w_d(vi)$ and a read from data item d returning v_i as $r_d(v)$.

- We assume that all data items have the null value, ⊥, at database initialization, and, unless otherwise specified, all transactions in the examples *commit*.

- In the invariant confluence analysis, we reason about *read-write histories* (or simply *histories*: sets of transactions consisting of read and write operations and their return values):

$$T_1: w_x(1) w_x(2)$$
$$T_2: w_x(3)$$
$$T_3: r_x(a)$$

Read Committed

- **Read Committed** ensures that users never read non-final writes to data
- For instance, in the example below, T_3 should never read $a = 1$, and, if T_2 aborts, T_3 should not read $a = 3$:

$$T_1: w_x(1) w_x(2)$$
$$T_2: w_x(3)$$
$$T_3: r_x(a)$$

- Read Committed is **invariant confluent**: if each individual user never observes non-final writes (i.e., each individual read-write history is valid under RC isolation), then their collective behavior (i.e., the “merged” histories) will not exhibit any reads of non-final writes
- A **coordination-free algorithm** for enforcing RC isolation: As a simple solution, clients can buffer their writes until they commit, or, alternatively, can send them to servers, who will not deliver their value to other readers until notified that the writes have been committed

Cut Isolation

- **Cut isolation** - if a transaction reads the same data more than once, it sees the same value each time.

- Under *CI* T_3 must read $a = 1$:

$$T_1: w_x(1)$$
$$T_2: w_x(2)$$
$$T_3: r_x(1) r_x(a)$$

- Item Cut Isolation is **invariant confluent**: if two histories are valid under Item Cut Isolation, unioning them will not change the return values of reads

- **Coordination-free implementation**: we can have transactions store a copy of any read data at the client such that the value that they read for each item never changes unless they overwrite it themselves

Monotonic Atomic View

- **Monotonic Atomic View (MAV) isolation** - either all or none of transactions' effects should succeed. Atomicity properties also restrict the updates visible to other transactions.

- Under MAV once some of the effects of a transaction T_i are observed by another transaction T_j, thereafter, all effects of T_i are observed by T_j. That is, if a transaction T_j reads a version of an object that transaction T_i wrote, then a later read by T_j cannot return a value whose later version is installed by T_i

- In the example below, under MAV, because T_2 has read T_1's write to y, T_2 must observe $b = c = 1$ (or later versions for each key):

$$T_1: w_x(1) w_y(1) w_z(1)$$
$$T_2: r_x(a) r_y(1) r_x(b) r_z(c)$$

- MAV is **invariant confluent** for read and write operations. If two histories obey MAV, then their union does not change the effects of the reads in each history.

Session guarantees

- A useful class of safety guarantees refer to real-time or client-centric ordering within a *session*, “an abstraction for the sequence of...operations performed during the execution of an application”

- Several session guarantees can be made with coordination-free execution
  - **The monotonic reads** guarantee requires that, within a session, subsequent reads to a given object “never return any previous values”; reads from each item progress according to a total order
  - **The monotonic writes** guarantee requires that each session’s writes become accessible to other transactions in the order they were committed.
  - **The writes follow reads** guarantee requires that, if a session observes an effect of transaction T_1 and subsequently commits transaction T_2, then another session can only observe effects of T_2 if it can also observe T_1’s effects (or later values that supersede T_1’s). Any order on transactions should respect this transitive order.
- The above guarantees are **invariant confluent** for reads and writes

Sticky availability

- the concept of **“stickiness**”: clients can ensure continuity between operations (e.g., reading their prior updates to a data item) by maintaining affinity or “stickiness” with a server or set of servers (a client can maintain stickiness by contacting the same server for each of its requests)

- Sticky availability permits three additional guarantees:
  - **Read your writes** requires that whenever a session reads a given data item d after writing a version d_i to it, the read returns the d_i or another version d_j, where $j > i$.
  - **PRAM (Pipelined Random Access Memory)** provides the illusion of serializing each of the operations (both reads and writes) within each session and is the combination of monotonic reads, monotonic writes, and read your writes
  - **Causal consistency** results from the combination of all session guarantees

Lost Update

- *Lost Update* - when one transaction T_1 reads a given data item, a second transaction T_2 updates the same data item, then T_1 modifies the data item based on its original read of the data item, “missing” or “losing” T_2’s newer update

- Consider a database containing only the following transactions:

$$T_1 : r_x(a)\, w_x\,(a + 2)$$
$$T_2 : w_x(2)$$

- If T_1 reads $a = 1$ but T_2’s write to x precedes T_1’s write operation, then the database will end up with $a = 3$, a state that could not have resulted in a serial execution due to T_2’s “Lost Update.”.

- It is impossible to prevent Lost Update in a highly available environment

- **Consistent Read, Snapshot Isolation (including Parallel Snapshot Isolation), and Cursor Stability** guarantees are all unavailable because they require preventing Lost Update phenomena.

Write Skew

- **Write Skew** is a generalization of Lost Update to multiple keys.
- It occurs when one transaction T_1 reads a given data item x, a second transaction T_2 reads a different data item y, then T_1 writes to y and commits and T_2 writes to x and commits.
- As an example of Write Skew, consider the following two transactions:

$$T_1 : r_y(0)\, w_x(1)$$
$$T_2 : r_x(0) w_y(1)$$

- If there was an integrity constraint between x and y such that only one of x or y should have value 1 at any given time, then this write skew would violate the constraint (which is preserved in serializable executions)
- As a generalization of Lost Update, Write Skew is also **unavailable to coordination-free** systems
- Repeatable Read and One-Copy Serializability need to prevent both Lost Update and Write Skew

Unachievable Recency Guarantees

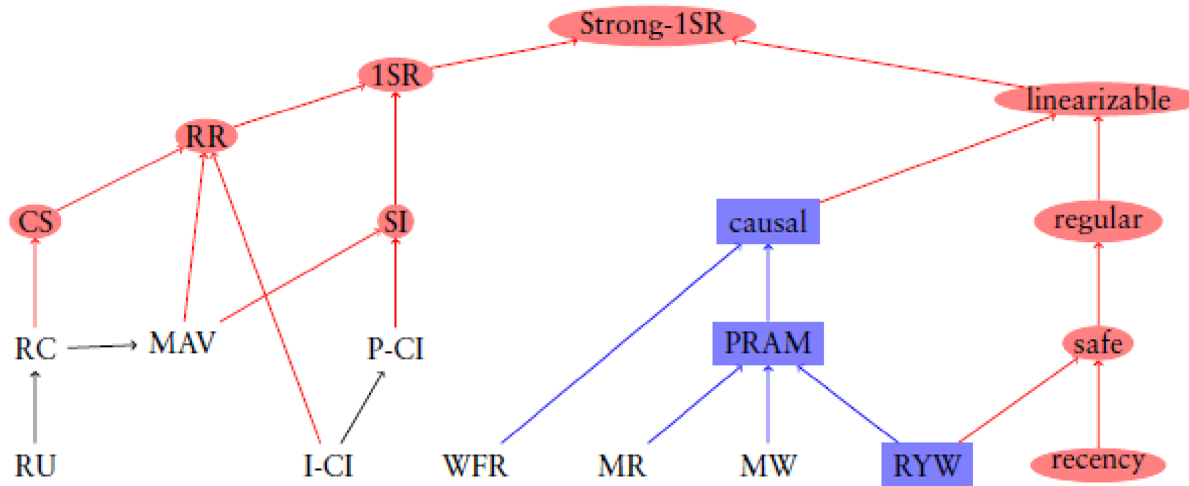
- Distributed data storage systems often make various **recency guarantees** on reads of data items
- Unfortunately, an indefinitely long partition can force an available system to violate any recency bound, so recency bounds are **not enforceable by coordination-free systems**
- One of the most famous of these guarantees is **linearizability**, which states that *reads will return the last completed write to a data item*. There are also several other (weaker) variants such as safe and regular register semantics.
- None of these guarantees are invariant confluent.

Summary of the considered models

Invariant Confluent	Read Uncommitted (RU), Read Committed (RC), Monotonic Atomic View (MAV), Item Cut Isolation (I-CI), Predicate Cut Isolation (P-CI), Writes Follow Reads (WFR), Monotonic Reads (MR), Monotonic Writes (MW)
Sticky	Read Your Writes (RYW), PRAM, Causal
Unavailable	Cursor Stability (CS)†, Snapshot Isolation (SI)†, Repeatable Read (RR)†‡, One-Copy Serializability (1SR)†‡, Recency⊕, Safe⊕, Regular⊕, Linearizability⊕, Strong 1SR†‡⊕

- Summary of invariant confluent, sticky, and non-invariant confluent models considered
- Non-invariant confluent models are labeled by cause: preventing lost update †, preventing write skew ‡, and requiring recency guarantees ⊕
- The resulting taxonomy is one of the first printed in the literature

Hierarchy of isolation models



- Partial ordering of invariant confluent, sticky (in boxes), and non-invariant confluent models (circled)
- Directed edges represent ordering by model strength

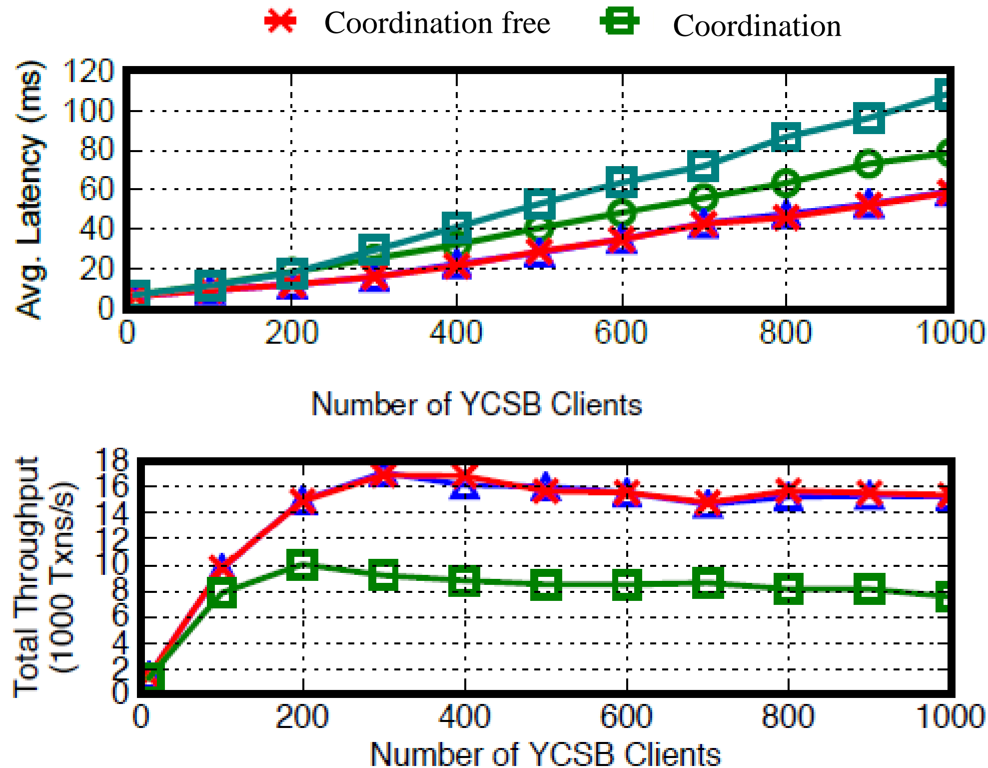
Database prototype

- To investigate the performance of coordination-free weak isolation implementation in a real-world environment, a database prototype was implemented
- **Goal** is *not* a complete performance analysis of coordination-free semantics but instead a demonstration of coordination-free designs on real-world infrastructure
- The database prototype is deployed in *clusters* across one or more datacenters and all clients stick within a datacenter to their respective cluster
- 5 Amazon Elastic Compute Cloud m1.xlarge instances (15GB RAM, with 4 cores comprising 8 “EC2 Compute Units”) are used as servers in each cluster
- To simulate a database workload the YCSB benchmark is used. Every eight YCSB operations from the default workload (50% reads, 50% writes) are grouped to form a transaction.

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

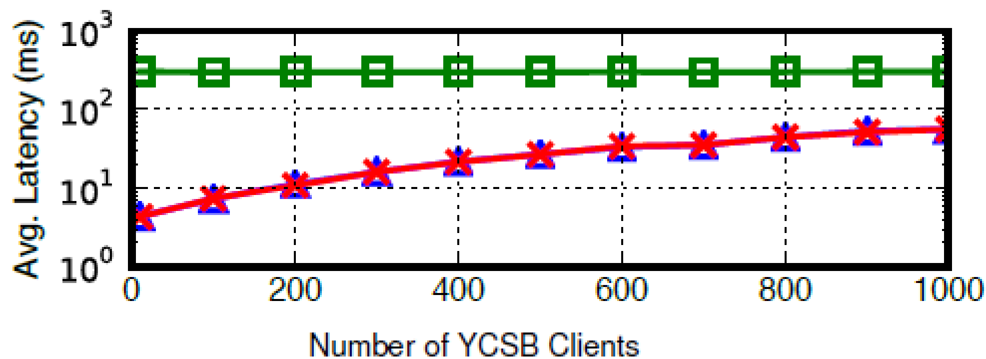
Experiment Results

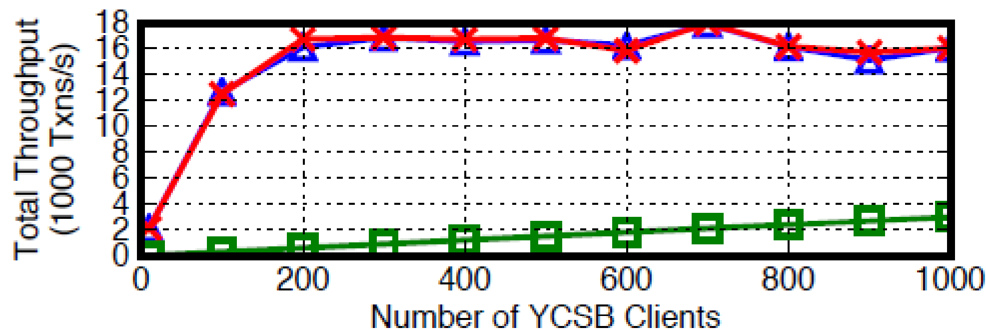
Figures show that, when operating two clusters within a single datacenter, coordinating each data item results in approximately half the throughput and double the latency of coordination-free configuration.



Experiment Results

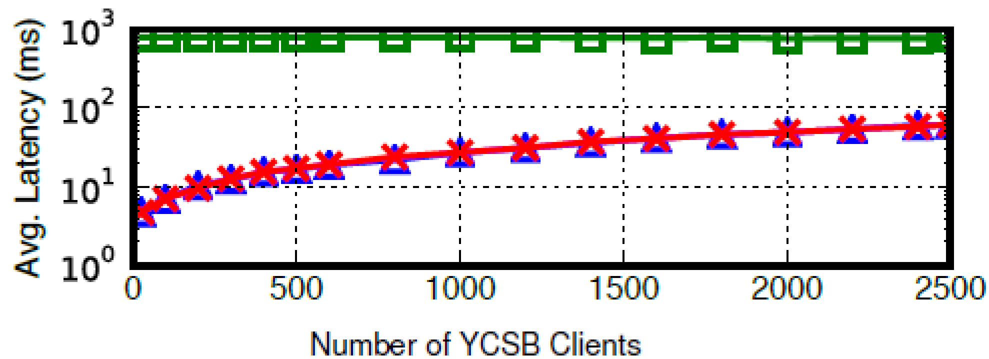
When the two clusters are deployed across the continental United States, the average latency of "coordination configuration" increases to 300ms. For the same number of YCSB client threads, "coordination configuration" has **substantially lower throughput than the coordination-free configurations**.

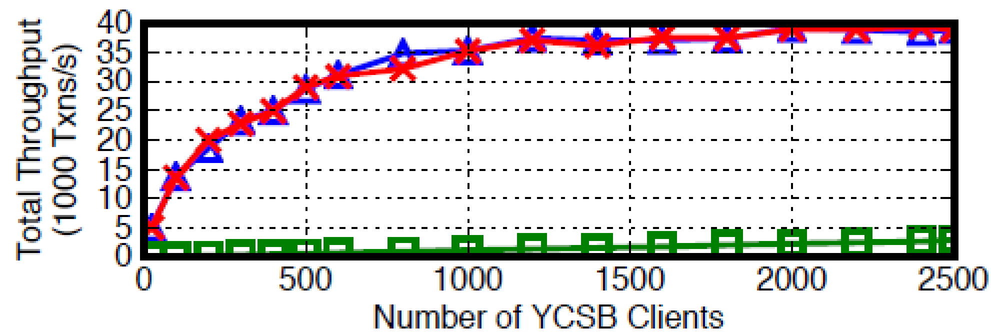




Experiment Results

When five clusters are deployed across the five EC2 datacenters, the trend continues: non coordination-free latency increases to nearly 800ms per transaction.





Conclusion

- It has been shown that many previously defined isolation and data consistency models are **invariant confluent** and can be implemented in a coordination-free manner

- A coordination-free database prototype was implemented that confirms that coordination free systems can provide useful semantics without substantial performance penalties

Спасибо за внимание!

101000, Россия, Москва, Мясницкая ул., д. 20
Тел.: (495) 621-7983, факс: (495) 628-7931
www.hse.ru