SCALABLE ATOMIC VISIBILITY WITH RAMP TRANSACTIONS

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica

UC Berkeley and University of Sydney
Overview deck with Cassandra discussion
@pbailis
NOSQL
NoSQL
NO SQL

DIDN'T WANT SQL
NOSQL

NO SQL

DIDN'T WANT SERIALIZABILITY
NOSQL
NO SQL

DIDN'T WANT SERIALIZABILITY
POOR PERFORMANCE
NOSQL
NO SQL

DIDN’T WANT SERIALIZABILITY
POOR PERFORMANCE
NOSQL
NO SQL
DIDN’T WANT SERIALIZABILITY
POOR PERFORMANCE
NOSQL

DIDN’T WANT SERIALIZABILITY

POOR PERFORMANCE

PEAK THROUGHPUT: 1/Delay

FOR CONTENDED OPERATIONS
NOSQL

DIDN’T WANT SERIALIZABILITY
POOR PERFORMANCE

PEAK THROUGHPUT: 1/DELAY
FOR CONTENTED OPERATIONS

at .5MS, 2K TXN/s
at 50MS, 20 TXN/s

DELAY
NOSQL

DIDN’T WANT SERIALIZABILITY
POOR PERFORMANCE
HIGH LATENCY
NO SQL
DIDN’T WANT SERIALIZABILITY
POOR PERFORMANCE
HIGH LATENCY
LIMITED AVAILABILITY
“NOT ONLY SQL”
STILL DON’T WANT SERIALIZABILITY
“NOT ONLY SQL”

STILL DON’T WANT SERIALIZABILITY
(DON’T WANT THE COSTS)
“NOT ONLY SQL”

STILL DON’T WANT SERIALIZABILITY
(DON’T WANT THE COSTS)
BUT WANT MORE FEATURES
"NOT ONLY SQL"

STILL DON’T WANT SERIALIZABILITY
(DON’T WANT THE COSTS)
BUT WANT MORE FEATURES

This paper!
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013
"TAO: Facebook's Distributed Data Store for the Social Graph"  
USENIX ATC 2013

**Denormalized Friend List**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Profile 1" /></td>
<td><img src="image2.png" alt="Profile 2" /></td>
<td><img src="image3.png" alt="Profile 3" /></td>
</tr>
<tr>
<td><img src="image4.png" alt="Profile 4" /></td>
<td><img src="image5.png" alt="Profile 5" /></td>
<td><img src="image6.png" alt="Profile 6" /></td>
</tr>
<tr>
<td><img src="image7.png" alt="Profile 7" /></td>
<td><img src="image8.png" alt="Profile 8" /></td>
<td><img src="image9.png" alt="Profile 9" /></td>
</tr>
</tbody>
</table>
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013

**Denormalized Friend List**

*Fast reads...*
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013

Denormalized Friend List

Fast reads...
...multi-entity updates
Denormalized Friend List

Fast reads...
...multi-entity updates

Add Friend
Denormalized Friend List

Fast reads...
...multi-entity updates

Add Friend
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013

Denormalized Friend List

Fast reads...
...multi-entity updates

Add Friend
“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013

**Denormalized Friend List**

- Fast reads...
- ...multi-entity updates
- Not cleanly partitionable
FOREIGN KEY DEPENDENCIES

“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013
FOREIGN KEY DEPENDENCIES

“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013

“On Brewing Fresh Espresso: LinkedIn’s Distributed Data Serving Platform” SIGMOD 2013
FOREIGN KEY DEPENDENCIES

“TAO: Facebook’s Distributed Data Store for the Social Graph”
USENIX ATC 2013

“On Brewing Fresh Espresso: LinkedIn’s Distributed Data Serving Platform”
SIGMOD 2013

“PNUTS: Yahoo!’s Hosted Data Serving Platform”
VLDB 2008
<table>
<thead>
<tr>
<th>ID</th>
<th>AGE</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>22</td>
<td><img src="image1.png" alt="Icon" /></td>
</tr>
<tr>
<td>412</td>
<td>72</td>
<td><img src="image2.png" alt="Icon" /></td>
</tr>
<tr>
<td>532</td>
<td>42</td>
<td><img src="image3.png" alt="Icon" /></td>
</tr>
<tr>
<td>892</td>
<td>13</td>
<td><img src="image4.png" alt="Icon" /></td>
</tr>
<tr>
<td>2345</td>
<td>1</td>
<td><img src="image5.png" alt="Icon" /></td>
</tr>
</tbody>
</table>
Partition by primary key (ID)

How should we look up by age?

ID: 123
AGE: 22

ID: 412
AGE: 72

ID: 532
AGE: 42

ID: 892
AGE: 13

ID: 2345
AGE: 1
SECONDARY INDEXING

Partition by primary key (ID)

How should we look up by age?
SECONDARY INDEXING

Partition by primary key (ID)

How should we look up by age?

Option I: Local Secondary Indexing
SECONDARY INDEXING

**Partition by primary key (ID)**

**How should we look up by age?**

Option I: Local Secondary Indexing
Build indexes co-located with primary data
SECONDARY INDEXING

Partition by primary key (ID)

How should we look up by age?

Option I: Local Secondary Indexing
Build indexes co-located with primary data
WRITE ONE SERVER, READ ALL
SECONDARY INDEXING

Partition by primary key (ID)

How should we look up by age?

Option I: Local Secondary Indexing
Build indexes co-located with primary data
WRITE ONE SERVER, READ ALL poor scalability
SECONDARY INDEXING

How should we look up by age?

Option I: Local Secondary Indexing
Build indexes co-located with primary data
WRITE ONE SERVER, READ ALL
poor scalability

Option II: Global Secondary Indexing
Partition indexes by secondary key

Partition by primary key (ID)

Partition by secondary attribute
SECONDARY INDEXING

How should we look up by age?

Option I: Local Secondary Indexing
Build indexes co-located with primary data

WRITE ONE SERVER, READ ALL
poor scalability

Option II: Global Secondary Indexing
Partition indexes by secondary key

WRITE 2+ SERVERS, READ ONE
How should we look up by age?

Option I: Local Secondary Indexing
Build indexes co-located with primary data
WRITE ONE SERVER, READ ALL

Option II: Global Secondary Indexing
Partition indexes by secondary key
WRITE 2+ SERVERS, READ ONE scalable lookups
How should we look up by age?

Option I: Local Secondary Indexing
Build indexes co-located with primary data
WRITE ONE SERVER, READ ALL
poor scalability

Option II: Global Secondary Indexing
Partition indexes by secondary key
WRITE 2+ SERVERS, READ ONE scalable lookups
SECONDARY INDEXING

How should we look up by age?

Option I: Local Secondary Indexing
Build indexes co-located with primary data
WRITE ONE SERVER, READ ALL

Option II: Global Secondary Indexing
Partition indexes by secondary key
WRITE 2+ SERVERS, READ ONE

poor scalability
scalable lookups
Life beyond Distributed Transactions: an Apostate’s Opinion
Position Paper

Pat Helland
Amazon.Com
705 Fifth Ave South
Seattle, WA 98104
USA
P.Helland@Amazon.com

The positions expressed in this paper are personal opinions and do not in any way reflect the positions of my employer, Amazon.com.

ABSTRACT

Many decades of work have been invested in the area of distributed transactions including protocols such as 2PC, Paxos, and various approaches to quorum. These protocols provide the application programmer a façade of global serializability. Personally, I have invested a non-trivial portion of my career as a strong advocate for the implementation and use of platforms providing guarantees of global serializability. My experience over the last decade has led me to liken these platforms to the Maginot Line. In general, application developers simply do not implement large scalable applications assuming distributed transactions. When they attempt to use distributed transactions, the projects founder because the performance costs and fragility make them impractical. Natural selection kicks in... Instead, applications are built using different techniques which do not provide the same transactional guarantees but still meet the needs of their businesses.

This paper explores and names some of the practical approaches used in the implementations of large-scale mission-critical applications in a world which rejects distributed transactions. We discuss the management of fine-grained pieces of application data which may be repartitioned over time as the application grows. We also discuss the design patterns used in sending messages between these repartitionable pieces of data.

The reason for starting this discussion is to raise awareness of new patterns for two reasons. First, it is my belief that this awareness can ease the challenges of people hand-crafting very large scalable applications. Second, by observing the patterns, hopefully the industry can work towards the creation of platforms that make it easier to build these very large applications.

1. INTRODUCTION

Let’s examine some goals for this paper, some assumptions that I am making for this discussion, and then some opinions derived from the assumptions. While I am keenly interested in high availability, this paper will ignore that issue and focus on scalability alone. In particular, we focus on the implications that fall out of assuming we cannot have large-scale distributed transactions.

Goals

This paper has three broad goals:

- Discuss Scalable Applications
  Many of the requirements for the design of scalable systems are understood implicitly by many application designers who build large systems.

1 The Maginot Line was a huge fortress that ran the length of the Franco-German border and was constructed at great expense between World War I and World War II. It successfully kept the German army from directly crossing the border between France and Germany. It was quickly bypassed by the Germans in 1940 who invaded through Belgium.

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/2.5/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

3rd Biennial Conference on Innovative DataSystems Research (CIDR) January 7-10, Asilomar, California USA.
ABSTRACT
Megastore is a storage system developed to meet the requirements of today’s interactive online services. Megastore blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS in a novel way, and provides both strong consistency guarantees and high availability. We provide fully serializable ACID semantics within fine-grained partitions of data. This partitioning allows us to synchronously replicate each write across a wide area network with reasonable latency and support seamless failover between datacenters. This paper describes Megastore’s semantics and replication algorithm. It also describes our experience supporting a wide range of Google production services built with Megastore.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Distributed databases; H.2.4 [Database Management]: Systems—concurrency, distributed databases

General Terms
Algorithms, Design, Performance, Reliability

Keywords
Large databases, Distributed transactions, Bigtable, Paxos

1. INTRODUCTION
Interactive online services are forcing the storage community to meet new demands as desktop applications migrate to the cloud. Services like email, collaborative documents, and social networking have been growing exponentially and are testing the limits of existing infrastructure. Meeting these services’ storage demands is challenging due to a number of conflicting requirements.

First, the Internet brings a huge audience of potential users, so the applications must be highly scalable. A service can be built rapidly using MySQL [10] as its datastore, but scaling the service to millions of users requires a complete redesign of its storage infrastructure. Second, services must compete for users. This requires rapid development of features and fast time-to-market. Third, the service must be responsive; hence, the storage system must have low latency. Fourth, the service should provide the user with a consistent view of its data—the result of an update should be visible immediately and durably. Seeing edits to a cloud-hosted spreadsheet vanish, however briefly, is a poor user experience. Finally, users have come to expect Internet services to be up 24/7, so the service must be highly available. The service must be resilient to many kinds of faults ranging from the failure of individual disks, machines, or routers all the way up to large-scale outages affecting entire datacenters.

These requirements are in conflict. Relational databases provide a rich set of features for easily building applications, but they are difficult to scale to hundreds of millions of users. NoSQL datastores such as Google’s Bigtable [10], Apache Hadoop’s HBase [1], or Facebook’s Cassandra [8] are highly scalable, but they lack strong consistency models. Composites complicate application development. Replicating data across datacenters while providing low latency is challenging, as is guaranteeing a consistent view of replicated data, especially during faults. Megastore is a storage system developed to meet the storage requirements of today’s interactive online services. It is novel in that it blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS. It uses synchronous replication to achieve high availability and a consistent view of the data. In brief, it provides fully serializable ACID semantics over distant replicas with low enough latencies to support interactive applications.

We accomplish this by taking a middle ground in the RDBMS vs. NoSQL design space: we partition the datastore and replicate each partition separately, providing full ACID semantics within partitions, but only limited consistency guarantees across them. We support traditional database features, such as secondary indexes, but only those features that can scale within user-tolerable latency limits, and only with the semantics that our partitioning scheme can support. We contend that the data for most Internet services can be suitably partitioned (e.g., by user) to make this approach viable, and that a small, but not sparse, set of features can substantially ease the burden of developing cloud applications.

Contrary to conventional wisdom [24, 28], we were able to
Life beyond Distributed Transactions: an Apostate’s Opinion
Position Paper

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

On Brewing Fresh Espresso: LinkedIn’s Distributed Data Serving Platform

Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradkar, Chris Beavers, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jagadish, Shi Lü, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramanian, Yun Sun, Sajid Topiwala, Cungen Tran, Jemiah Westerman, David Zhang
LinkedIn, Inc.
Mountain View, CA, USA
{lqiao,ksurlaker,sdas,tquiggle,bsschulman,bghosh,acurtis,oseeliger,zzhang,aauradkar,cbeavers,gbrandt,mgandhi,kgopalakrishna,wip,sjagadish,slu,apache,rramesh,asebastian,rshanbhag,ssubramanian,ysun,sttopiwala,cran,jwesterman,dzhang}@linkedin.com

ABSTRACT

Espresso is a document-oriented distributed data serving platform that has been built to address LinkedIn’s requirements for a scalable, performant, source-of-truth primary store. It provides a hierarchical document model, transactional support for modifications to related documents, real-time secondary indexing, on-the-fly schema evolution and provides a timeline consistent change capture stream. This paper describes the motivation and design principles involved in building Espresso, the data model and capabilities exposed to clients, details of the replication and secondary indexing implementation and presents a set of experimental results that characterize the performance of the system along various dimensions.

When we set out to build Espresso, we chose to apply best practices in industry, already published works in research and our own internal experience with different consistency models. Along the way, we built a novel generic distributed cluster management framework, a partition-aware change-capture pipeline and a high-performance inverted index implementation.

Categories and Subject Descriptors: C.2.4 [Distributed Systems]: Distributed databases; H.2.4 [Database Management]: Systems-concurrency, distributed databases

General Terms: Algorithms, Design, Performance, Reliability

Keywords: Large Databases, Transactions, Secondary Indexing, Cluster Management, Change Data Capture, MySQL

1. INTRODUCTION

To meet the needs of online applications, Relational Database Management Systems (RDBMSs) have been developed and deployed widely, providing support for data schema, rich transactions, and enterprise scale.

In its early days, the LinkedIn data ecosystem was quite simple. A single RDBMS contained a handful of tables for user data such as profiles, connections, etc. This RDBMS was augmented with two specialized systems: one provided full text search of the corpus of user profile data, the other provided efficient traversal of the relationship graph. These latter two systems were kept up-to-date by Datahub [14], a change capture stream that propagates writes to the RDBMS primary data store, in commit order, to the search and graph clusters.

Over the years, as LinkedIn evolved, so did its data needs. LinkedIn now provides a diverse offering of products and services to over 200 million members worldwide, as well as a comprehensive set of tools for our Talent Solutions and Marketing Solutions businesses. The early pattern of a primary, strongly consistent, data store that accepts reads and writes, then generates a change capture stream to fulfill online and offline processing requirements, has become a common design pattern. Many, if not most, of the primary data requirements of LinkedIn do not require the full functionality of a RDBMS; nor can they justify the associated costs.

Using RDBMS technology has some associated pain points. First, the existing RDBMS installation requires costly, specialized hardware and extensive caching to meet scale and latency requirements. Second, adding capacity requires a long planning cycle, and is difficult to do at scale with 100% uptime. Third, product agility introduces a new set of challenges for schemas and their management. Often the data models don’t scale to the specific form and
Life beyond Distributed Transactions: an Apostate’s Opinion
Position Paper

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

On Brewing Fresh Espresso: LinkedIn’s Distributed Data Serving Platform

PNUTS: Yahoo!’s Hosted Data Serving Platform

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerrinen
Yahoo! Research

ABSTRACT

We describe PNUTS, a massively parallel and geographically distributed database system for Yahoo!’s web applications. PNUTS provides data storage organized as hashed or ordered tables, low latency for large numbers of concurrent requests including updates and queries, and novel per-record consistency guarantees. It is a hosted, centrally managed, and geographically distributed service, and utilizes automated load-balancing and failover to reduce operational complexity. The first version of the system is currently serving in production. We describe the motivation for PNUTS and the design and implementation of its table storage and replication layers, and then present experimental results.

1. INTRODUCTION

Modern web applications present unprecedented data management challenges, even for relatively “simple” tasks like managing session state, content meta-data, and user-generated content such as tags and comments. The foremost requirements of a web application are availability, consistently good response time for geographically dispersed users, and high availability. At the same time, web applications can frequently tolerate relaxed consistency guarantees. We now examine these requirements in more detail.

Scalability. For popular applications such as Flickr and del.icio.us, the need for a scalable data engine is obvious [4]. We want not only architectural scalability, but the ability to scale during periods of rapid growth by adding resources with minimal operational effort and minimal impact on system performance.

Yahoo!’s internal SLAs for page load time, placing stringent response time requirements on the data management platform. Given that web users are scattered across the globe, it is critical to have data replicas on multiple continents for low-latency access. Consider social network applications—alumni of a university in India may reside in North America and Europe as well as Asia, and a particular user’s data may be accessed both by the user from his home in London as well as by his friends in Mumbai and San Francisco. Ideally, the data platform should guarantee fast response times to geographically distributed users, even under rapidly changing load conditions brought on by flash crowds, denial of service attacks, etc.

High Availability and Fault Tolerance. Yahoo! applications must provide a high degree of availability, with application-specific trade-offs in the degree of fault-tolerance required and the degree of consistency that is deemed acceptable in the presence of faults; e.g., all applications want to be able to read data in the presence of failures, which some insist on also being able to write in the presence of failures, even at the cost of risking some data consistency. Downtime means money is lost. If we cannot serve ads, Yahoo! does not get paid; if we cannot render pages, we disappoint users. Thus service must continue in the face of a variety of failures including server failures, network partitions and the loss of power in a co-location facility.

Relaxed Consistency Guarantees. Traditional database systems have long provided us with a well-understood model for reasoning about consistency in the presence of concurrent operations, namely serializable transactions [5]. However, there is a tradeoff between performance and availability on one hand, and between availability and consistency on the other.
Global Secondary Indexes

Topics

- Attribute Projections
- Creating a Global Secondary Index
- Querying a Global Secondary Index
- Data Synchronization Between Tables and Global Secondary Indexes
- Provisioned Throughput Considerations for Global Secondary Indexes
- Storage Considerations for Global Secondary Indexes
- Guidelines for Global Secondary Indexes
- Working with Global Secondary Indexes Using the AWS SDK for Java Low-Level API
- Working with Global Secondary Indexes Using the AWS SDK for .NET Low-Level API
- Working with Global Secondary Indexes Using the AWS SDK for PHP Low-Level API

Some applications might need to perform many kinds of queries, using a variety of different attributes as query criteria. To support these requirements, you can create one or more global secondary indexes and issue Query requests against these indexes. To illustrate, consider a table named GameScores that keeps track of users and scores for a mobile gaming application. Each item in GameScores is identified by a hash key (UserId) and a range key (GameTitle). The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown)

GameScores

<table>
<thead>
<tr>
<th>UserId</th>
<th>GameTitle</th>
<th>TopScore</th>
<th>TopScoreDateTime</th>
<th>Wins</th>
<th>Losses</th>
</tr>
</thead>
<tbody>
<tr>
<td>(hash key)</td>
<td>(range key)</td>
<td>TopScore</td>
<td>TopScoreDateTime</td>
<td>Wins</td>
<td>Losses</td>
</tr>
</tbody>
</table>
SECONDARY INDEXING

Partition by primary key (ID)

How should we look up by age?

Option I: Local Secondary Indexing
Build indexes co-located with primary data

Option II: Global Secondary Indexing
Partition indexes by secondary key

Partition by secondary attribute

scalable lookups
poor scalability

WRITE 2+ SERVERS, READ ONE
WRITE ONE SERVER, READ ALL

OVERVIEW

INCONSISTENT
GLOBAL 2ı

(INCONSISTENT)

GLOBAL 2ı

(PROPOSED)

GLOBAL 2ı

GLOBAL 2ı
TABLE:
ALL USERS
TABLE:
ALL USERS
TABLE: ALL USERS

TABLE: USERS OVER 25
MATERIALIZED VIEWS

TABLE: ALL USERS

TABLE: USERS OVER 25

RELEVANT RECENT EXAMPLES IN GOOGLE PERCOLATOR TWITTER RAINBIRD LINKEDIN ESPRESSO PAPERS
HOW SHOULD WE CORRECTLY MAINTAIN
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXES
MATERIALIZED VIEWS
SNAPSHOT ISOLATION

REPEATABLE READ (PL-2.99)

CURSOR STABILITY

READ COMMITTED

READ UNCOMMITTED

SERIALIZABILITY

SNAPSHOT ISOLATION

LINEARIZABILITY

CAUSAL

PRAM

RYW

EVENTUAL CONSISTENCY
SERIALIZABILITY

REPEATABLE READ (PL-2.99)

SNAPSHOT ISOLATION

CURSOR STABILITY

LINEARIZABILITY

READ COMMITTED

CAUSAL

READ UNCOMMITTED

PRAM

EVENTUAL CONSISTENCY

RYW

MANY

SUFFICIENT
REPEATABLE READ (PL-2.99)
CURSOR STABILITY
SERIALIZABILITY
SNAPSHOT ISOLATION
LINEARIZABILITY

REQUIRE SYNCHRONOUS COORDINATION

READ COMMITTED
READ UNCOMMITTED
EVENTUAL CONSISTENCY

CAUSAL
PRAM
RYW

MANY
SUFFICIENT
SERIALIZABILITY

REPEATABLE READ (PL-2.99)

SNAPSHOT ISOLATION

CURSOR STABILITY

LINEARIZABILITY

REQUIRE SYNCHRONOUS COORDINATION

COORDINATION-FREE

READ COMMITTED

READ UNCOMMITTED

CAUSAL

PRAM

RYW

EVENTUAL CONSISTENCY

MANY SUFFICIENT
SERIALIZABILITY

SNAPSHOT ISOLATION

LINEARIZABILITY

REPEATABLE READ
(PL-2.99)

CURSOR STABILITY

REQUIRE SYNCHRONOUS COORDINATION

COORDINATION-FREE

Facebook TAO
LinkedIn Espresso
Yahoo! PNUTS
Google Megastore
Google App Engine
Twitter Rainbird
Amazon DynamoDB

CONSCIOUS CHOICES!
SERIALIZABILITY
SNAPSHOT ISOLATION
LINEARIZABILITY
REPEATABLE READ (PL-2.99)
CURSOR STABILITY
REQUIRE SYNCHRONOUS COORDINATION
COORDINATION-FREE
SUFFICIENT
READ COMMITTED
CAUSAL
READ UNCOMMITTED
PRAM
RYW
EVENTUAL CONSISTENCY
COORDINATION-FREE
SUFFICIENT
SERIALIZABILITY

REPEATABLE READ (PL-2.99)
CURSOR STABILITY

SNAPSHOT ISOLATION
LINEARIZABILITY

REQUIRE SYNCHRONOUS COORDINATION

COORDINATION-FREE

RAMP (THIS PAPER)

SUFFICIENT
INSUFFICIENT

READ COMMITTED
CAUSAL
PRAM
RYW

READ UNCOMMITTED
EVENTUAL CONSISTENCY
READ

ATOMIC

MULTI-

PARTITION

TRANSACTIONS
RAMP TRANSACTIONS

EFFICIENTLY MAINTAIN
RAMP TRANSACTIONS EFFICIENTLY MAINTAIN FOREIGN KEY DEPENDENCIES
RAMP TRANSACTIONS EFFICIENTLY MAINTAIN FOREIGN KEY DEPENDENCIES SECONDARY INDEXES
RAMP TRANSACTIONS
EFFICIENTLY MAINTAIN
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXES
MATERIALIZED VIEWS
RAMP TRANSACTIONS
EFFICIENTLY MAINTAIN
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXES
MATERIALIZED VIEWS

BY PROVIDING
ATOMIC VISIBILITY
ATOMIC VISIBILITY
ATOMIC VISIBILITY

Informally:
Either all of each transaction’s updates are visible, or none are
Informally:
Either all of each transaction’s updates are visible, or none are.
ATOMIC VISIBILITY

Informally:
Either all of each transaction’s updates are visible, or none are.

WRITE X = 1
WRITE Y = 1
ATOMIC VISIBILITY

Informally:
Either all of each transaction’s updates are visible, or none are.

WRITE X = 1
WRITE Y = 1
READ X = 1
READ Y = 1
Informally:
Either all of each transaction’s updates are visible, or none are.

**ATOMIC VISIBILITY**

- **WRITE X = 1**
- **WRITE Y = 1**

- **READ X = 1**
- **READ Y = 1**

**OR**
Informally:
Either all of each transaction’s updates are visible, or none are.

- **WRITE X = 1**
- **WRITE Y = 1**
- **READ X = 1**
- **READ Y = 1**

OR

- **READ X = ∅**
- **READ Y = ∅**
ATOMIC VISIBILITY

Informally:
Either all of each transaction’s updates are visible, or none are.

READ X = 1
READ Y = 1

OR

READ X = ∅
READ Y = ∅
ATOMIC VISIBILITY

Informally:
Either all of each transaction’s updates are visible, or none are.

- READ X = 1
- READ Y = 1

OR

- READ X = Ø
- READ Y = Ø

BUT NOT

- READ X = 1
- READ Y = Ø
Informally:
Either all of each transaction’s updates are visible, or none are:

**ATOMIC VISIBILITY**

- READ X = 1
- READ Y = 1
  - OR
  - READ X = Ø
  - READ Y = Ø

**BUT NOT**

- READ X = 1
- READ Y = Ø
  - OR
  - READ X = Ø
  - READ Y = 1
Formally:
A transaction $T_j$ exhibits fractured reads if transaction $T_i$ writes versions $x_m$ and $y_n$ (in any order, with $x$ possibly but not necessarily equal to $y$), $T_j$ reads version $x_m$ and version $y_k$, and $k < n$.

A system provides Read Atomic isolation (RA) if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data.

We also provide per-item PRAM guarantees with per-transaction regular semantics (see paper Appendix)
RAMP TRANSACTIONS
GUARANTEE
ATOMIC VISIBILITY
RAMP TRANSACTIONS
GUARANTEE
ATOMIC VISIBILITY
WHILE ENSURING
RAMP TRANSACTIONS GUARANTEE ATOMIC VISIBILITY WHILE ENSURING PARTITION INDEPENDENCE
RAMP TRANSACTIONS
GUARANTEE
ATOMIC VISIBILITY
WHILE ENSURING
PARTITION INDEPENDENCE
clients only access servers responsible for data in transactions
RAMP TRANSACTIONS
GUARANTEE
ATOMIC VISIBILITY

WHILE ENSURING
PARTITION INDEPENDENCE

clients only access servers responsible for data in transactions
RAMP TRANSACTIONS GUARANTEE ATOMIC VISIBILITY

WHILE ENSURING PARTITION INDEPENDENCE

clients only access servers responsible for data in transactions

\[ W(X=1) \]

\[ W(Y=1) \]
RAMP TRANSACTIONS

GUARANTEE ATOMIC VISIBILITY

WHILE ENSURING PARTITION INDEPENDENCE
clients only access servers responsible for data in transactions

AND SYNCHRONIZATION INDEPENDENCE
transactions always commit and no client can cause another client to block
RAMP TRANSACTIONS

GUARANTEE

ATOMIC VISIBILITY

ARE NOT SERIALIZABLE

DO NOT PREVENT LOST UPDATE

DO NOT PREVENT WRITE SKEW

ALLOW CONCURRENT UPDATES
RAMP TRANSACTIONS
GUARANTEE
ATOMIC VISIBILITY

ARE NOT SERIALIZABLE
DO NOT PREVENT LOST UPDATE
DO NOT PREVENT WRITE SKEW
ALLOW CONCURRENT UPDATES

ARE GUIDED BY REAL WORLD USE CASES
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING
MATERIALIZED VIEWS
RAMP TRANSACTIONS

GUARANTEE ATOMIC VISIBILITY

ARE NOT SERIALIZABLE
DO NOT PREVENT LOST UPDATE
DO NOT PREVENT WRITE SKEW
ALLOW CONCURRENT UPDATES

ARE GUIDED BY REAL WORLD USE CASES

FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING
MATERIALIZED VIEWS

Facebook TAO
LinkedIn Espresso
Yahoo! PNUTS
Google Megastore
Google App Engine
Twitter Rainbird
Amazon DynamoDB
STRAWMAN: LOCKING

\[ X = O \quad Y = O \]
STRAWMAN: LOCKING

X = O

Y = O

W(X = 1)

W(Y = 1)
STRAWMAN: LOCKING

X = 0

Y = 0

W(X = 1)

W(Y = 1)
STRAWMAN: LOCKING

X = 0

Y = 0

W(X=1)

W(Y=1)
STRAWMAN: LOCKING

X = 1
Y = 1

W(X=1)
W(Y=1)
STRAWMAN: LOCKING

X = 1

Y = 1

W(X=1)

W(Y=1)
STRAWMAN: LOCKING

X = 1

Y = 1

W(X=1)

W(Y=1)
STRAWMAN: LOCKING

X=1

Y=1

W(X=1)

W(Y=1)
STRAWMAN: LOCKING

\[ X = 1 \]
\[ Y = 1 \]

\[ W(X=1) \]
\[ W(Y=1) \]
\[ R(X=1) \]
STRAWMAN: LOCKING

X=1

Y=1

W(X=1)
W(Y=1)
R(X=1)
R(Y=1)
STRAWMAN: LOCKING

W(X=1)
W(Y=1)
STRAWMAN: LOCKING

W(X=1)

W(Y=1)
STRAWMAN: LOCKING

X = 1

Y = 0

W(X=1)

W(Y=1)

R(X=?)
STRAWMAN: LOCKING

X = 1

Y = 0

W(X=1)  R(X=?)
W(Y=1)  R(Y=?)
STRAWMAN: LOCKING

\[ X = 1 \]
\[ Y = 0 \]

W(\(X=1\))  R(\(X=?\))  W(\(Y=1\))  R(\(Y=?\))

ATOMIC VISIBILITY COUPLED WITH MUTUAL EXCLUSION
STRAWMAN: LOCKING

ATOMIC VISIBILITY COUPLED WITH MUTUAL EXCLUSION
STRAWMAN: LOCKING

\[ X=1 \]

\[ W(X=1) \]

\[ W(Y=1) \]

\[ Y=0 \]

\[ R(X=?), R(Y=?) \]

ATOMIC VISIBILITY COUPLED WITH MUTUAL EXCLUSION
STRAWMAN: LOCKING

X = 1

W(X=1)  R(X=?)

W(Y=1)  R(Y=?)

Y = 0

ATOMIC VISIBILITY COUPLED WITH MUTUAL EXCLUSION
STRAWMAN: LOCKING

ATOMIC VISIBILITY COUPLED WITH MUTUAL EXCLUSION

RTT unavailability!
STRAWMAN: LOCKING

SIMILAR ISSUES IN MVCC,
(global timestamp assignment/application)
SERIALIZABLE OCC,
(multi-partition validation, liveness)
PRE-SCHEDULING
(scheduling, multi-partition execution)

ATOMIC VISIBILITY COUPLED WITH MUTUAL EXCLUSION

W(X=1)  R(X=?)
W(Y=1)  R(Y=?)
STRAWMAN: LOCKING

FUNDAMENTAL TO "STRONG" SEMANTICS

W(X=1)  R(X=?)
W(Y=1)  R(Y=?)

SIMILAR ISSUES IN MVCC,
(global timestamp assignment/application)
SERIALIZABLE OCC,
(multi-partition validation, liveness)
PRE-SCHEDULING
(scheduling, multi-partition execution)

ATOMIC VISIBILITY COUPLED WITH MUTUAL EXCLUSION
BASIC IDEA

\[ W(X=1) \quad W(Y=1) \quad Y=0 \quad R(X=?), \quad R(Y=?) \]

\[ X=1 \]

\[ W(X=1) \quad W(Y=1) \quad Y=0 \quad R(X=?), \quad R(Y=?) \]
BASIC IDEA

X=1
W(X=1)
W(Y=1)

Y=0
R(X=?)
R(Y=?)
BASIC IDEA

LET CLIENTS RACE, but HAVE READERS “CLEAN UP”

\( X=1 \)

\( W(X=1) \)
\( W(Y=1) \)

\( Y=0 \)

\( R(X=?) \)
\( R(Y=?) \)
BASIC IDEA

LET CLIENTS RACE, but HAVE READERS “CLEAN UP”

METADATA

X=1

Y=0

W(X=1)

W(Y=1)

R(X=?)

R(Y=?)
BASIC IDEA

LET CLIENTS RACE, but HAVE READERS “CLEAN UP”

METADATA + LIMITED MULTI-VERSIONING

W(X=1)
W(Y=1)
R(X=?)
R(Y=?)
BASIC IDEA

LET CLIENTS RACE, but HAVE READERS "CLEAN UP"

METADATA + LIMITED MULTI-VERSIONING

FOR NOW: READ-ONLY, WRITE-ONLY TXNS
RAMP-Fast

last committed stamp for x: 0

last committed stamp for y: 0
RAMP-Fast

last committed stamp for x: 0

known versions of x

last committed stamp for y: 0

known versions of y
### RAMP-Fast

**last committed stamp for x: 0**

**known versions of x**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

**last committed stamp for y: 0**

**known versions of y**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>
RAMP-Fast

<table>
<thead>
<tr>
<th>last committed stamp for x: 0</th>
</tr>
</thead>
</table>

**known versions of x**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>last committed stamp for y: 0</th>
</tr>
</thead>
</table>

**known versions of y**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>
**RAMP-Fast**

- **last committed stamp for x: 0**

  **known versions of x**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

- **last committed stamp for y: 0**

  **known versions of y**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>
### RAMP-Fast

#### last committed stamp for x: 0

#### known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

#### last committed stamp for y: 0

#### known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>
RAMP-Fast

last committed stamp for x: 0

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>
RAMP-Fast

last committed stamp for x: 0

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>
**RAMP-Fast**

**last committed stamp for x: 0**

**known versions of x**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

**last committed stamp for y: 0**

**known versions of y**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>
RAMP-Fast

1.) Assign unique (logical) transaction timestamp.

- e.g., time concat client ID concat sequence number

- timestamp: 242
RAMP-Fast

last committed stamp for x: 0

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
</tbody>
</table>

W(X=1)  
W(Y=1)

timestamp: 242

1.) Assign unique (logical) transaction timestamp.
RAMP-Fast

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
1.) Assign unique (logical) transaction timestamp.

2.) Add write to known versions on partition.
RAMP-Fast

1.) Assign unique (logical) transaction timestamp.

2.) Add write to known versions on partition.
# RAMP-Fast

**last committed stamp for x: 0**

## known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**last committed stamp for y: 0**

## known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.

2.) Add write to known versions on partition.

3.) Commit and update last committed stamp.
last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.
1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

last committed stamp for y: 242

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

last committed stamp for y: 242

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.

2.) Add write to known versions on partition.

3.) Commit and update last committed stamp.
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

last committed stamp for y: 242

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.
**RAMP-Fast**

**last committed stamp for x: 242**

**known versions of x**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**last committed stamp for y: 0**

**known versions of y**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.

1. W(X=1)
2. W(Y=1)
3. R(X=?)
4. R(Y=?)
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

timestamp: 242

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

timestamp: 242

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.
RAMP-Fast

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.

RACE!!!
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.

RACE!!!
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.

2.) Add write to known versions on partition.

3.) Commit and update last committed stamp.

W(X=1)

W(Y=1)
timestamp: 242

RACE!!!

R(X=1)

R(Y=0)
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.

timestamp: 242

RACE!!!
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.

RACE!!!
# RAMP-Fast

### Known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

- **Last committed stamp for x:** 242

### Known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- **Last committed stamp for y:** 0

---

**Record the items written in the transaction**

1. Assign unique (logical) transaction timestamp.
2. Add write to known versions on partition.
3. Commit and update last committed stamp.

---

**RACE!!!**

- R(X=1)
- R(Y=0)
RAMP-Fast

**known versions of x**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

**known versions of y**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

**last committed stamp for x:** 242

**last committed stamp for y:** 0

1.) Assign unique (logical) transaction timestamp.
2.) Add write to known versions on partition.
3.) Commit and update last committed stamp.

**RECORD THE ITEMS WRITTEN IN THE TRANSACTION**

RACE!!!
RAMP-Fast

**known versions of x**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

**last committed stamp for x**: 242

**known versions of y**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

**last committed stamp for y**: 0

R(X=?)

R(Y=?)
**RAMP-Fast**

- **last committed stamp for x**: 242

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

- **last committed stamp for y**: 0

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

- R(X=?)
- R(Y=?)
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

R(X=?)

R(Y=?)
1.) Read last committed:

**RAMP-Fast**

**last committed stamp for x: 242**

**known versions of x**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

**last committed stamp for y: 0**

**known versions of y**

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

R(X=?)

R(Y=?)
**RAMP-Fast**

### Known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

last committed stamp for x: 242

### Known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

1.) Read last committed:

\[X=1 @ 242, \{Y\}\]
### known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

### known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

1.) Read last committed:

- X = 1 @ 242, {Y}
- Y = NULL @ 0, {}
**RAMP-Fast**

---

### Known versions of X

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

Last committed stamp for X: 242

---

### Known versions of Y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

Last committed stamp for Y: 0

---

1. Read last committed:
   - X = 1 @ 242, {Y}
   - Y = NULL @ 0, {}

2. Calculate missing versions:
RAMP-Fast

1.) Read last committed:
   \[ X=1 @ 242, \{Y\} \]
   \[ Y=\text{NULL} @ 0, \{\} \]

2.) Calculate missing versions:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>HIGHEST TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>242</td>
</tr>
<tr>
<td>Y</td>
<td>242</td>
</tr>
</tbody>
</table>
last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

1.) Read last committed:

X=1 @ 242, {y}
Y=NULL @ 0, {}

2.) Calculate missing versions:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>HIGHEST TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>242</td>
</tr>
<tr>
<td>Y</td>
<td>242</td>
</tr>
</tbody>
</table>
RAMP-Fast

1.) Read last committed:
   - X=1 @ 242, {Y}
   - Y=NULL @ 0, {}

2.) Calculate missing versions:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>HIGHEST TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>242</td>
</tr>
<tr>
<td>Y</td>
<td>242</td>
</tr>
</tbody>
</table>

3.) Fetch missing versions.
**RAMP-Fast**

### Known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

Last committed stamp for x: 242

### Known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

Last committed stamp for y: 0

1.) Read last committed:

- X=1 @ 242, {Y}
- Y=NULL @ 0, {}

2.) Calculate missing versions:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>HIGHEST TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>242</td>
</tr>
<tr>
<td>Y</td>
<td>242</td>
</tr>
</tbody>
</table>

3.) Fetch missing versions.

- Y=1 @ 242, {X}

*(Send required timestamp in request)*
RAMP-Fast

last committed stamp for x: 242

known versions of x

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

last committed stamp for y: 0

known versions of y

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{x}</td>
</tr>
</tbody>
</table>

1.) Read last committed:
   - X=1 @ 242, \{y\}
   - Y=NULL @ 0, {}  

2.) Calculate missing versions:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>HIGHEST TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>242</td>
</tr>
<tr>
<td>Y</td>
<td>242</td>
</tr>
</tbody>
</table>

3.) Fetch missing versions.
   - Y=1 @ 242, \{X\}  
   (Send required timestamp in request)
RAMP-Fast

**last committed stamp for x:** 242

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>VALUE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>{}</td>
</tr>
<tr>
<td>242</td>
<td>1</td>
<td>{y}</td>
</tr>
</tbody>
</table>

**last committed stamp for y:** 0

<table>
<thead>
<tr>
<th>ITEM</th>
<th>HIGHEST TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>242</td>
</tr>
<tr>
<td>Y</td>
<td>242</td>
</tr>
</tbody>
</table>

1.) Read last committed:
   - X=1 @ 242, {Y}
   - Y=NULL @ 0, {}

2.) Calculate missing versions:

3.) Fetch missing versions.
   - Y=1 @ 242, {X}

(Send required timestamp in request)

4.) Return resulting set.

2PC ENSURES NO WAIT AT SERVER
RAMP-Fast
RAMP-Fast

2 RTT writes:
RAMP-Fast

2 RTT writes:

2PC, without blocking synchronization
RAMP-Fast

2 RTT writes:
- 2PC, without blocking synchronization

ENSURES READERS NEVER WAIT!
RAMP-Fast

2 RTT writes:
2PC, without blocking synchronization
metadata size linear in transaction size

ENSURES READERS NEVER WAIT!
RAMP-Fast

2 RTT writes:
2PC, without blocking synchronization
metadata size linear in transaction size

1 RTT reads:
in race-free case

ENSURES READERS NEVER WAIT!
RAMP-Fast

2 RTT writes:
2PC, without blocking synchronization
metadata size linear in transaction size

1 RTT reads:
in race-free case

2 RTT reads:
otherwise

ENSURES READERS NEVER WAIT!
RAMP-Fast

2 RTT writes:
   2PC, without blocking synchronization
   metadata size linear in transaction size

1 RTT reads:
   in race-free case

2 RTT reads:
   otherwise

no fast-path synchronization

ENSURES READERS NEVER WAIT!
RAMP-Fast

2 RTT writes:
- 2PC, without blocking synchronization
- Metadata size linear in transaction size

1 RTT reads:
- In race-free case

2 RTT reads:
- Otherwise

No fast-path synchronization

Ensures readers never wait!

Can we use less metadata?
RAMP-Small

2 RTT writes:
RAMP-Small

2 RTT writes:

- same basic protocol as RAMP-Fast
- but drop all RAMP-Fast metadata
RAMP-Small

2 RTT writes:
  same basic protocol as RAMP-Fast
  but drop all RAMP-Fast metadata

2 RTT reads
RAMP-Small

2 RTT writes:
same basic protocol as RAMP-Fast
but drop all RAMP-Fast metadata

2 RTT reads ← always
RAMP-Small

2 RTT writes:
- same basic protocol as RAMP-Fast
- but drop all RAMP-Fast metadata

2 RTT reads ← always

1.) For each item, fetch the highest committed timestamp.
2.) Request highest matching write with timestamp in step 1.
RAMP-Small

2 RTT writes:
  same basic protocol as RAMP-Fast
  but drop all RAMP-Fast metadata

2 RTT reads ← always

1.) For each item, fetch the highest committed timestamp.
2.) Request highest matching write with timestamp in step 1.

INTUITION:
RAMP-Small

2 RTT writes:
   same basic protocol as RAMP-Fast
   but drop all RAMP-Fast metadata

2 RTT reads ← always

1.) For each item, fetch the highest committed timestamp.
2.) Request highest matching write with timestamp in step 1.

INTUITION:
X → time 523
RAMP-Small

2 RTT writes:
same basic protocol as RAMP-Fast
but drop all RAMP-Fast metadata

2 RTT reads $\leftarrow$ always

1.) For each item, fetch the highest committed timestamp.
2.) Request highest matching write with timestamp in step 1.

INTUITION:
X $\rightarrow$ time 523
Y $\rightarrow$ time 247
RAMP-Small

2 RTT writes:
same basic protocol as RAMP-Fast
but drop all RAMP-Fast metadata

2 RTT reads \(\text{always}\)

1.) For each item, fetch the highest committed timestamp.
2.) Request highest matching write with timestamp in step 1.

INTUITION:
\begin{align*}
X & \rightarrow \text{time 523} \\
Y & \rightarrow \text{time 247} \\
Z & \rightarrow \text{time 842}
\end{align*}
RAMP-Small

2 RTT writes:
 same basic protocol as RAMP-Fast
 but drop all RAMP-Fast metadata

2 RTT reads ← always

1.) For each item, fetch the highest committed timestamp.
2.) Request highest matching write with timestamp in step 1.

INTUITION:
X → time 523
Y → time 247
Z → time 842

\{247, 523, 842\}
RAMP-Small

2 RTT writes:
same basic protocol as RAMP-Fast
but drop all RAMP-Fast metadata

2 RTT reads ← always

1.) For each item, fetch the highest committed timestamp.
2.) Request highest matching write with timestamp in step 1.

INTUITION:
X → time 523
Y → time 247
Z → time 842

\{247, 523, 842\}
partial commits will be in this set
RAMP-Small

2 RTT writes:
same basic protocol as RAMP-Fast
but drop all RAMP-Fast metadata

2 RTT reads \overset{\text{always}}{\leftarrow}

1.) For each item, fetch the highest committed timestamp.
2.) Request highest matching write with timestamp in step 1.

INTUITION:

X $\rightarrow$ time 523
Y $\rightarrow$ time 247
Z $\rightarrow$ time 842

\{247, 523, 842\}

partial commits will be in this set
send it to all participating servers
# RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>Read RTT (best case)</th>
<th>Read RTT (worst case)</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1)</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+ε</td>
<td>2</td>
<td>O(1)</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Write RTT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm</td>
<td>Write RTT</td>
<td>READ RTT (best case)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-----------</td>
<td>----------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>2+\epsilon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- RAMP-Fast: 2 cycles for write set summary, 1 cycle (best case) read, 2 cycles (worst case) read.
- RAMP-Small: 2 cycles, 2 cycles for timestamp, 2 cycles for Bloom filter.
## RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+(\varepsilon)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+(\varepsilon)</td>
<td>2</td>
<td>(O(\log(2)))</td>
</tr>
</tbody>
</table>
# RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len) write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1) timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+(\varepsilon)</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
## RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len) write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1) timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+(\varepsilon)</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**Bloom Filter Summarizes Write Set False Positives: Extra RTTs**
## RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>$O(txn \text{ len})$ write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>$O(1)$ timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>$1+\varepsilon$</td>
<td>2</td>
<td>$O\left(\frac{[txn \text{ len}]^*\log(1/\varepsilon)}{\log(2)^2}\right)$</td>
</tr>
</tbody>
</table>

**BLOOM FILTER SUMMARIZES WRITE SET FALSE POSITIVES: EXTRA RTTs**
# RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>$O(\text{txn len})$ write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>$O(1)$ timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+$\varepsilon$</td>
<td>2</td>
<td>$O(B)$ Bloom filter</td>
</tr>
</tbody>
</table>

Note: $O(B)$ denotes the time complexity of the Bloom filter operation.
## RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>(O(txn \text{ len})) write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>(O(1)) timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+(\varepsilon)</td>
<td>2</td>
<td>(O(B)) Bloom filter</td>
</tr>
</tbody>
</table>

- **AVOID IN-PLACE UPDATES**
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+ε</td>
<td>2</td>
<td>O(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bloom filter</td>
</tr>
</tbody>
</table>

- AVOID IN-PLACE UPDATES
- EMBRACE RACES TO IMPROVE CONCURRENCY
## RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len) write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1) timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+ε</td>
<td>2</td>
<td>O(B) Bloom filter</td>
</tr>
</tbody>
</table>

- AVOID IN-PLACE UPDATES
- EMBRACE RACES TO IMPROVE CONCURRENCY
- ALLOW READERS TO REPAIR PARTIAL WRITES
## RAMP Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1 + $\varepsilon$</td>
<td>2</td>
<td>O(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bloom filter</td>
</tr>
</tbody>
</table>

- **AVOID IN-PLACE UPDATES**
- **EMBRACE RACES TO IMPROVE CONCURRENCY**
- **ALLOW READERS TO REPAIR PARTIAL WRITES**
- **USE 2PC TO AVOID READER STALLS**
Additional Details
Garbage collection:
- limit read transaction duration to $K$ seconds
- GC overwritten versions after $K$ seconds
**Additional Details**

**Garbage collection:**
- limit read transaction duration to $K$ seconds
- GC overwritten versions after $K$ seconds

**Replication**
- paper assumes linearizable masters
- extendable to “AP” systems
- see HAT by Bailis et al., VLDB 2014
Additional Details

Garbage collection:
- limit read transaction duration to K seconds
- GC overwritten versions after K seconds

Replication
- paper assumes linearizable masters
- extendable to “AP” systems
- see HAT by Bailis et al., VLDB 2014

Failure handling:
- blocked 2PC rounds do not block clients
- stalled commits? versions are not GC’d
- if desirable, use CTP termination protocol
## RAMP PERFORMANCE

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len) write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1) timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+(\varepsilon)</td>
<td>2</td>
<td>O(B) Bloom filter</td>
</tr>
</tbody>
</table>
## RAMP PERFORMANCE

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+(\epsilon)</td>
<td>2</td>
<td>O(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bloom filter</td>
</tr>
</tbody>
</table>

Evaluated on EC2 cr1.8xlarge instances

(cluster size: 1-100 servers; default: 5)

Open sourced on GitHub; see link at end of talk
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

Doesn't provide atomic visibility

No Concurrency Control
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

Throughput (txn/s) vs. Concurrent Clients for No Concurrency Control.
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

The diagram shows the throughput (txn/s) plotted against the number of concurrent clients. The x-axis represents the number of concurrent clients, ranging from 0 to 10,000. The y-axis represents the throughput, ranging from 0 to 180K.

The graph compares different concurrency control mechanisms:
- No Concurrency Control
- Serializable 2PL

The y-axis is labeled as Throughput (txn/s), and the x-axis is labeled as Concurrent Clients.
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

The graph shows the throughput (transactions per second) plotted against the number of concurrent clients. The x-axis represents the number of concurrent clients, ranging from 0 to 10,000, while the y-axis represents the throughput in transactions per second, ranging from 0 to 180,000.

The graph includes the following concurrency control strategies:
- **No Concurrency Control**
- **Serializable 2PL**
- **Write Locks Only**

Different symbols and line styles are used to distinguish between these strategies.

Key observations:
- **No Concurrency Control** shows a steady increase in throughput as the number of concurrent clients increases.
- **Serializable 2PL** maintains a relatively stable throughput, indicating that it is less affected by increasing concurrency.
- **Write Locks Only** also shows a steady increase in throughput, similar to **No Concurrency Control**, suggesting that write locks are not a significant bottleneck under these conditions.

Overall, the graph suggests that increasing the number of concurrent clients leads to a proportional increase in throughput for both **No Concurrency Control** and **Write Locks Only**, while **Serializable 2PL** remains relatively constant, indicating its effectiveness in managing concurrency for read operations.
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

![Graph showing throughput for different concurrency control methods.]

- **No Concurrency Control**
- **Serializable 2PL**
- **Write Locks Only**

Also doesn’t provide atomic visibility.
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

Throughput (txn/s) vs. Concurrent Clients

- No Concurrency Control
- Serializable 2PL
- Write Locks Only

Representative of coordinated approaches
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

[Graph showing throughput (txn/s) vs. concurrent clients for different concurrency control strategies, including No Concurrency Control, Serializable 2PL, and Write Locks Only.]
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

The image shows a graph with the x-axis labeled "Concurrent Clients" ranging from 0 to 10,000 and the y-axis labeled "Throughput (txn/s)" ranging from 0 to 180,000. The graph compares different concurrency control methods under the YCSB WorkloadA with 95% reads, 1 million items, and 4 items per transaction.

- **No Concurrency Control** (triangle): shows the throughput without any concurrency control mechanisms.
- **RAMP-Fast** (diamond): represents the throughput with the RAMP-Fast concurrency control method.
- **Serializable 2PL** (square): illustrates the throughput using the Serializable 2PL protocol.
- **Write Locks Only** (cross): demonstrates the throughput when write locks are used exclusively.

Each method is represented by a distinct marker, and the graph provides a clear comparison of throughput across different concurrency control strategies as the number of concurrent clients increases.
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

Within ~5% of baseline
Latency in paper (comparable)
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

The diagram illustrates the throughput (transactions per second) as a function of the number of concurrent clients for different concurrency control strategies. The strategies include:

- **No Concurrency Control**
- **RAMP-Fast**
- **Serializable 2PL**
- **Write Locks Only**

The graph shows that RAMP-Fast generally outperforms the other strategies, especially at higher concurrency levels.
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

The graph shows the throughput (txn/s) as a function of the number of concurrent clients for various concurrency control strategies:

- **No Concurrency Control**
- **RAMP-Fast**
- **Serializable 2PL**
- **RAMP-Small**
- **Write Locks Only**

The x-axis represents the number of concurrent clients, ranging from 0 to 10,000. The y-axis represents the throughput in transactions per second (txn/s), ranging from 0 to 180,000.

Different strategies show varying performance levels, with RAMP-Fast generally outperforming others, especially at higher concurrency levels.
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

Throughput (txn/s)

Concurrent Clients

- No Concurrency Control
- RAMP-Fast
- Serializable 2PL
- RAMP-Small
- Write Locks Only

Always needs 2 RTT reads
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

Throughput (txn/s)

Concurrent Clients

- No Concurrency Control
- RAMP-Fast
- Serializable 2PL
- RAMP-Small
- Write Locks Only
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn

The graph shows the throughput (txn/s) for different concurrency control mechanisms as a function of the number of concurrent clients. The mechanisms compared are:

- No Concurrency Control
- RAMP-Fast
- Serializable 2PL
- RAMP-Small
- Write Locks Only
- RAMP-Hybrid

The graph indicates that RAMP-Fast and RAMP-Hybrid provide the highest throughput, followed by Serializable 2PL, RAMP-Small, and Write Locks Only. No Concurrency Control has the lowest throughput.
YCSB: WorkloadA, 95% reads, 1M items, 4 items/txn
YCSB: WorkloadA, 1M items, 4 items/txn, 5K clients

Throughput (txn/s) vs. Percentage Reads graph.
YCSB: WorkloadA, 1M items, 4 items/txn, 5K clients

No Concurrency Control
YCSB: WorkloadA, 1M items, 4 items/txn, 5K clients

Linear scaling; due to 2RTT writes, races
YCSB: WorkloadA, 1M items, 4 items/txn, 5K clients

Percentage Reads

Throughput (txn/s)

- No Concurrency Control
- RAMP-Fast
- Serializable 2PL
- RAMP-Small
- Write Locks Only
- RAMP-Hybrid

Graph showing the performance of different concurrency control methods with varying percentage of reads.
YCSB: uniform access, 1M items, 4 items/txn, 95% reads
YCSB: uniform access, 1M items, 4 items/txn, 95% reads

Throughput (ops/s) vs. Number of Servers

- No Concurrency Control
YCSB: uniform access, 1M items, 4 items/txn, 95% reads

The graph shows the throughput (ops/s) as a function of the number of servers for different concurrency control mechanisms.

- **No Concurrency Control**
- **RAMP-Fast**
- **RAMP-Small**
- **RAMP-Hybrid**

The graph demonstrates that RAMP-Hybrid has the highest throughput, followed by RAMP-Fast, RAMP-Small, and No Concurrency Control.
YCSB: uniform access, 1M items, 4 items/txn, 95% reads

![Graph showing performance comparison between different concurrency control methods.](image)
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>O(txn len) write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>O(1) timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+ε</td>
<td>2</td>
<td>O(B) Bloom filter</td>
</tr>
</tbody>
</table>

## RAMP Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write RTT</th>
<th>READ RTT (best case)</th>
<th>READ RTT (worst case)</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP-Fast</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>(O(txn\ len)) write set summary</td>
</tr>
<tr>
<td>RAMP-Small</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>(O(1)) timestamp</td>
</tr>
<tr>
<td>RAMP-Hybrid</td>
<td>2</td>
<td>1+(\varepsilon)</td>
<td>2</td>
<td>(O(B)) Bloom filter</td>
</tr>
</tbody>
</table>

More results in paper: Transaction length, contention, value size, latency, failures
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING
MATERIALIZED VIEWS
### How RAMP Handles:

- **Foreign Key Dependencies**
- **Secondary Indexing**
- **Materialized Views**
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES

SECONDARY INDEXING
MATERIALIZED VIEWS
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES

MULTI-PUT
(DELETES VIA TOMBSTONES)

SECONDARY INDEXING
MATERIALIZED VIEWS
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING

MATERIALIZED VIEWS
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING
MATERIALIZED VIEWS
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING
MATERIALIZED VIEWS
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING

Maintain list of matching record IDs and versions
e.g., HAS_BEARD={52@512, 412@52, 123@512}
merge lists on commit/read (LWW by timestamp for conflicts)

MATERIALIZED VIEWS
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING

Maintain list of matching record IDs and versions
e.g., HAS_BEARD={52@512, 412@52, 123@512}
merge lists on commit/read (LWW by timestamp for conflicts)

LOOKUPs: READ INDEX, THEN FETCH DATA

MATERIALIZED VIEWS
HOW RAMP HANDLES:
FOREIGN KEY DEPENDENCIES
SECONDARY INDEXING

Maintain list of matching record IDs and versions
e.g., HAS_BEARD={52@512, 412@52, 123@512}
merge lists on commit/read (LWW by timestamp for conflicts)

LOOKUPs: READ INDEX, THEN FETCH DATA

MATERIALIZED VIEWS

SIMILAR FOR SELECT/PROJECT
SERIALIZABILITY

REPEATABLE READ

SNAPSHOT ISOLATION

LINEARIZABILITY

CURSOR STABILITY

REQUIRE SYNCHRONOUS COORDINATION

COORDINATION-FREE

READ COMMITTED

CAUSAL

READ UNCOMMITTED

PRAM

EVENTUAL CONSISTENCY

RYW

COORDINATION-FREE

SUFFICIENT

INSUFFICIENT
SERIALIZABILITY
- REPEATABLE READ (PL-2.99)
- CURSOR STABILITY
- SNAPSHOT ISOLATION
- LINEARIZABILITY

REQUIRE SYNCHRONOUS COORDINATION

COORDINATION-FREE

SUFFICIENT
- EVENTUAL CONSISTENCY

INSUFFICIENT
- READ COMMITTED
- READ UNCOMMITTED
- CAUSAL
  - PRAM
  - RYW
- EVENTUAL CONSISTENCY
SERIALIZABILITY
  REPEATABLE READ (PL-2.99)
  CURSOR STABILITY

SNAPSHOT ISOLATION

LINEARIZABILITY

REQUIRE SYNCHRONOUS COORDINATION

COORDINATION-FREE

ATOMIC VISIBILITY VIA RAMP

READ COMMITTED

READ UNCOMMITTED

CAUSAL
  PRAM
  RYW

EVENTUAL CONSISTENCY

VIA RAMP

SUFFICIENT

INSUFFICIENT
RAMP IN CASSANDRA
RAMP IN CASSANDRA
USES
REQUIREMENTS
RAMP IN CASSANDRA

USES

REQUIREMENTS

IMPLEMENTATION
RAMP IN CASSANDRA

STRAIGHTFORWARD USES:

• Add atomic visibility to atomic batch operations
• Expose as CQL isolation level
  • USING CONSISTENCY READ_ATOMIC
  • Encourage use in multi-put, multi-get
• Treat as basis for global secondary indexing
  • CREATE GLOBAL INDEX on users (age )
RAMP IN CASSANDRA

REQUIREMENTS:

• Unique timestamp generation for transactions
  • Use node ID from ring
  • Other form of UUID
  • Hash transaction contents*

• Limited multi-versioning for prepared and old values
  • RAMP doesn’t actually require true MVCC
  • One proposal: keep a look aside cache
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:
Lookaside cache for prepared and old values

Standard C* Table stores
last committed write

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>335</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1240</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1402</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2201</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:
Lookaside cache for prepared and old values

Standard C* Table stores last committed write

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>335</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1240</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1402</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2201</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shadow table stores prepared-but-not-committed and overwritten versions

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>335</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2201</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:
Lookaside cache for prepared and old values

<table>
<thead>
<tr>
<th>Standard C* Table stores last committed write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>335</td>
</tr>
<tr>
<td>1240</td>
</tr>
<tr>
<td>1402</td>
</tr>
<tr>
<td>2201</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shadow table stores prepared-but-not-committed and overwritten versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
<tr>
<td>335</td>
</tr>
<tr>
<td>2201</td>
</tr>
</tbody>
</table>

Transparent to end-users
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:
Lookaside cache for prepared and old values

<table>
<thead>
<tr>
<th>Standard C* Table stores last committed write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>335</td>
</tr>
<tr>
<td>1240</td>
</tr>
<tr>
<td>1402</td>
</tr>
<tr>
<td>2201</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shadow table stores prepared-but-not-committed and overwritten versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
<tr>
<td>335</td>
</tr>
<tr>
<td>2201</td>
</tr>
</tbody>
</table>

Overwritten versions have TTL set to max read transaction time, do not need durability.

Transparent to end-users.
RAMP IN CASSANDRA
POSSIBLE IMPLEMENTATION:
RAMP IN CASSANDRA
POSSIBLE IMPLEMENTATION:

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CONSISTENCY LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Prepare</td>
<td>CL.QUORUM</td>
</tr>
<tr>
<td>Write Commit</td>
<td>CL.ANY or higher</td>
</tr>
<tr>
<td>First-round Read</td>
<td>CL.ANY/CL.ONE</td>
</tr>
<tr>
<td>Second-round Read</td>
<td>CL.QUORUM</td>
</tr>
</tbody>
</table>
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CONSISTENCY LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Prepare</td>
<td>CL.QUORUM</td>
</tr>
<tr>
<td>Write Commit</td>
<td>CL.ANY or higher</td>
</tr>
<tr>
<td>First-round Read</td>
<td>CL.ANY/CL.ONE</td>
</tr>
<tr>
<td>Second-round Read</td>
<td>CL.QUORUM</td>
</tr>
</tbody>
</table>

To avoid stalling, second-round reads must be able to access prepared writes.
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CONSISTENCY LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Prepare</td>
<td>CL.QUORUM</td>
</tr>
<tr>
<td>Write Commit</td>
<td>CL.ANY or higher</td>
</tr>
<tr>
<td>First-round Read</td>
<td>CL.ANY/CL.ONE</td>
</tr>
<tr>
<td>Second-round Read</td>
<td>CL.QUORUM</td>
</tr>
</tbody>
</table>

To avoid stalling, second-round reads must be able to access prepared writes.
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CONSISTENCY LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Prepare</td>
<td>CL.QUORUM</td>
</tr>
<tr>
<td>Write Commit</td>
<td>CL.ANY or higher</td>
</tr>
<tr>
<td>First-round Read</td>
<td>CL.ANY/CL.ONE</td>
</tr>
<tr>
<td>Second-round Read</td>
<td>CL.QUORUM</td>
</tr>
</tbody>
</table>

To avoid stalling, second-round reads must be able to access prepared writes.
RAMP IN CASSANDRA
POSSIBLE IMPLEMENTATION:
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:

DC1
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:

DC1

DC2
RAMP IN CASSANDRA

POSSIBLE IMPLEMENTATION:

Run algorithms on a per-DC basis, with use of CL.LOCAL_QUORUM instead of full CL.QUORUM
Global indexes

Details
- Type: New Feature
- Priority: Major
- Component/s: API, Core
- Labels: None
- Status: OPEN
- Resolution: Unresolved
- Fix Version/s: 3.0

Description
Local indexes are suitable for low-cardinality data, where spreading the index across the cluster is a Good Thing. However, for high-cardinality data, local indexes require querying most nodes in the cluster even if only a handful of rows is returned.

Activity
Jonathan Ellis added a comment - 11/Dec/13 23:16

Most application-maintained indexes solve this problem by denormalizing the base table row into the index entry. The problem is that this means we can't do lazy updates of the index; we need to keep the index perfectly (or, "eventually perfectly") in sync with the base table. Which in turns means we need to linearize updates to an indexed table. That was a performance hit but otherwise reasonable when we did that for local indexes; for partitioned indexes it's not
RAMP TRANSACTIONS:
RAMP TRANSACTIONS:

• Provide atomic visibility, as required for maintaining FKs, scalable indexing, mat views
RAMP TRANSACTIONS:

• Provide atomic visibility, as required for maintaining FKs, scalable indexing, mat views

• Avoid in-place updates, mutual exclusion, any synchronous/blocking coordination
RAMP TRANSACTIONS:

• Provide atomic visibility, as required for maintaining FKs, scalable indexing, mat views

• Avoid in-place updates, mutual exclusion, any synchronous/blocking coordination

• Use metadata with limited multi versioning, reads repair partial writes
RAMP TRANSACTIONS:

• Provide atomic visibility, as required for maintaining FKs, scalable indexing, mat views

• Avoid in-place updates, mutual exclusion, any synchronous/blocking coordination

• Use metadata with limited multi versioning, reads repair partial writes

• 1-2RTT overhead, pay only during contention
RAMP TRANSACTIONS:

- Provide atomic visibility, as required for maintaining FKs, scalable indexing, mat views
- Avoid in-place updates, mutual exclusion, any synchronous/blocking coordination
- Use metadata with limited multi versioning, reads repair partial writes
- 1-2RTT overhead, pay only during contention

Thanks!

http://tiny.cc/ramp-code
http://tiny.cc/ramp-intro
@pbailis
IMAGE/FONT CREDITS
Punk designed by my name is mud from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Queen designed by Bohdan Burmich from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Guy Fawkes designed by Anisha Varghese from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Emperor designed by Simon Child from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Baby designed by Les vieux garçons from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Baby designed by Les vieux garçons from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Gandhi designed by Luis Martins from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Database designed by Anton Outkine from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Girl designed by Rodrigo Vidinich from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Child designed by Gemma Garner from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Customer Service designed by Veysel Kara from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Punk Rocker designed by Simon Child from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Pyramid designed by misirlou from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Person designed by Stefania Bonacasa from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Record designed by Diogo Trindade from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Window designed by Juan Pablo Bravo from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Balloon designed by Julien Deveaux from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Balloon designed by Julien Deveaux from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Crying designed by Megan Sheehan from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Sad designed by Megan Sheehan from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Happy designed by Megan Sheehan from the Noun Project Creative Commons – Attribution (CC BY 3.0)
Happy designed by Megan Sheehan from the Noun Project Creative Commons – Attribution (CC BY 3.0)
User designed by JM Waideaswaran from the Noun Project Creative Commons – Attribution (CC BY 3.0)

COCOGOOSE font by ZetaFonts COMMON CREATIVE NON COMMERCIAL USE