Apache Cassandra

A decentralized, massively scalable data store.

L. Molari[1]

[1]Università di Bologna, Seconda Facoltà di Ingegneria

January 11, 2013

Outline

Outline

CAP Theorem

Consistency, Availability, Partition tolerance: Pick 2

CAP Theorem - Brewer, E.

A distributed system cannot simultaneously provide more than two out of the following three guarantees:

- Consistency
- Availability
- Partition tolerance

CAP Theorem

Consistency, Availability, Partition tolerance: Pick 2

CAP Theorem - Brewer, E.

A distributed system cannot simultaneously provide more than two out of the following three guarantees:

- Consistency
- Availability
- Partition tolerance

CAP Theorem

Consistency, Availability, Partition tolerance: Pick 2

CAP Theorem - Brewer, E.

A distributed system cannot simultaneously provide more than two out of the following three guarantees:

- Consistency
- Availability
- Partition tolerance

CAP Theorem

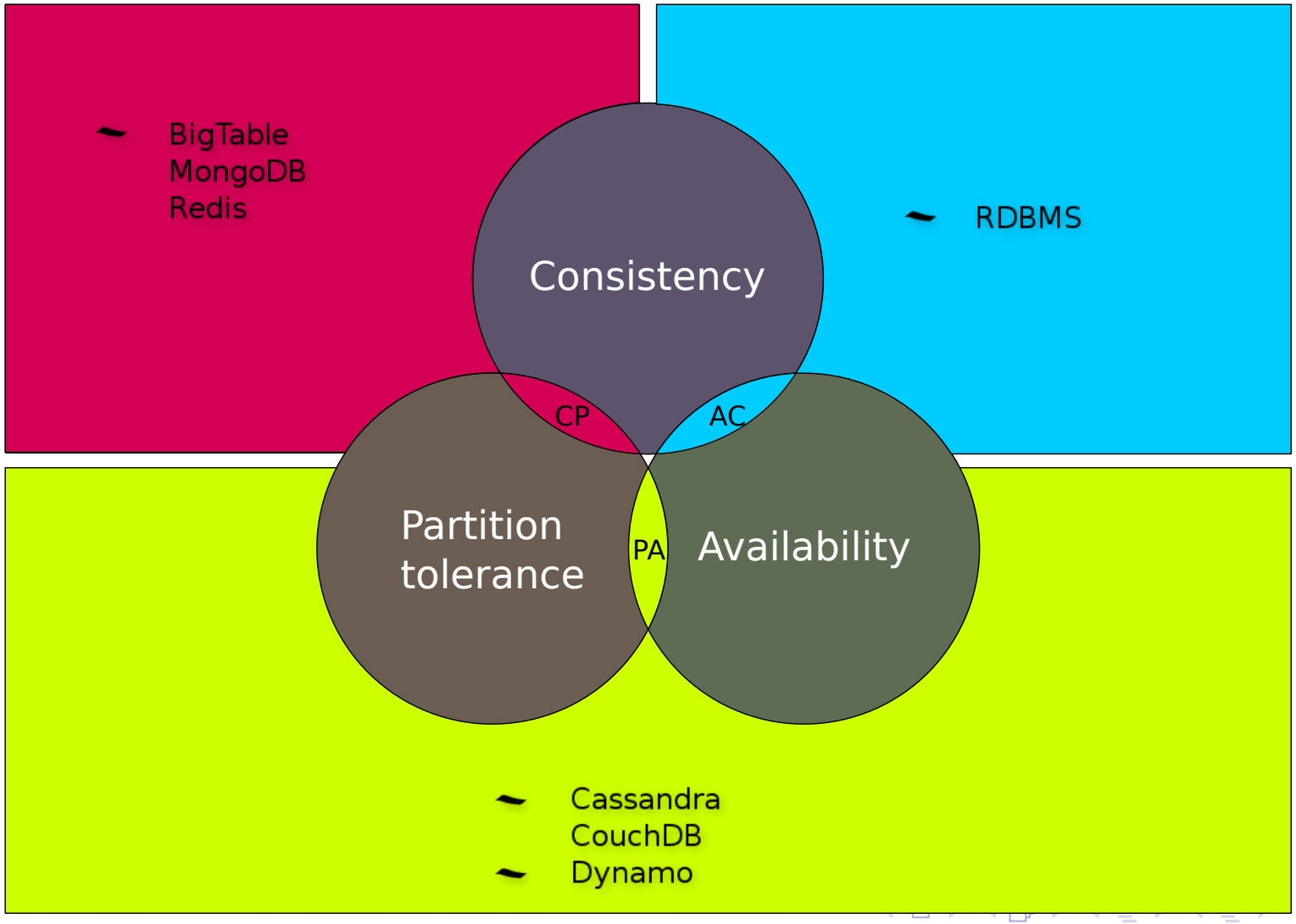
Consistency, Availability, Partition tolerance: Pick 2

CAP Theorem - Brewer, E.

A distributed system cannot simultaneously provide more than two out of the following three guarantees:

- Consistency
- Availability
- Partition tolerance

CAP Theorem


BigTable
MongoDB
Redis
RDBMS
Consistency
CP
AC
Partition tolerance
PA
Availability
Cassandra
CouchDB
Dynamo

CAP Theorem
Interpreting the theorem

Partitions?

- In distributed systems, partitions are unavoidable.
- Machine failures can be seen as a kind of network partitions.

Think CAP as $P \implies A$ XOR C:

- As partition occurrs, the tradeoff is between:
  - Availability, at the cost of inconsistency.
  - Consistency, at the cost of downtime.

CAP Theorem

Interpreting the theorem

Partitions?

- In distributed systems, partitions are unavoidable.
- Machine failures can be seen as a kind of network partitions.

Think CAP as $P \implies A$ XOR C:

- As partition occurrs, the tradeoff is between:
  - Availability, at the cost of inconsistency.
  - Consistency, at the cost of downtime.

Variation over CAP Theorem

A better classification for real-life systems

Insight

If there's no partition, the tradeoff is about how much Latency the system employs to achieve a certain degree of Consistency.

Variation - Abadi, D.

A better system classification is $P?(A|C : L|C)$:

- On Partition, the tradeoff is between Availability and Consistency.
- Otherwise, the tradeoff is between Latency and Consistency.

Variation over CAP Theorem

A better classification for real-life systems

Insight

If there's no partition, the tradeoff is about how much Latency the system employs to achieve a certain degree of Consistency.

Variation - Abadi, D.

A better system classification is $P?(A|C : L|C)$:

- On Partition, the tradeoff is between Availability and Consistency.
- Otherwise, the tradeoff is between Latency and Consistency.

Variation over CAP Theorem

Envisioning the classification

Systems following $P?C$ usually follow : C.

- $P?C : C$ (e.g. fully ACID systems, traditional RDBMS).
- $P?A : C$ give up on Consistency as partitions occurr.
- $P?A : L$ high availability systems focused on low latency and high scalability.
- **Yahoo's PNUTS *(Sherpa)*** is $P?C : L$ *(yes!)*: relaxes Consistency on normal operation, but doesn't give up any additional Consistency during partitions.

Variation over CAP Theorem

Envisioning the classification

Systems following $P?C$ usually follow : C.

- $P?C : C$ (e.g. fully ACID systems, traditional RDBMS).
- $P?A : C$ give up on Consistency as partitions occurr.
- $P?A : L$ high availability systems focused on low latency and high scalability.
- **Yahoo's PNUTS *(Sherpa)*** is $P?C : L$ *(yes!)*: relaxes Consistency on normal operation, but doesn't give up any additional Consistency during partitions.

ACID - BASE

ACID: gives up on Availability

Atomic	doesn't break *indivisible* consistency (*"all or nothing"*).
Consistent	committed updates don't break domain constraints.
Isolated	doesn't break *serializable* consistency.
Durable	committed updates are persistent.

ACID - BASE

BASE: gives up on Consistency

Basically Available availability is the main focus of the system.

Soft state state of the system can change anytime, even without inputs.

Eventually consistent system will become consistent after a long enough timeframe without inputs.

Outline

Two Phase Commit

Facts

- Is a distributed algorithm taking place in a Distributed Atomic Transaction (DAT).
- Coordinates the processes that participates in a DAT.
- Is a type of Consensus protocol.
- Tolerates a wide fraction of failure configurations.

Two Phase Commit: Execution

Phase 1: COMMIT Request phase

- A COORDINATOR send a QUERY TO COMMIT to each PARTICIPANT of the transaction, and waits for all replies.
- Each PARTICIPANT:
  - performs the transaction up to the point in which he's asked to COMMIT.
  - replies with a message:
    - AGREEMENT (if he would COMMIT the transaction).
    - ABORT (if he would ROLLBACK the transaction).

Two Phase Commit: Execution

Phase 1: COMMIT Request phase

- A COORDINATOR send a QUERY TO COMMIT to each PARTICIPANT of the transaction, and waits for all replies.
- Each PARTICIPANT:
  - performs the transaction up to the point in which he's asked to COMMIT.
  - replies with a message:
    - AGREEMENT (if he would COMMIT the transaction).
    - ABORT (if he would ROLLBACK the transaction).

Two Phase Commit: Execution

Phase 1: COMMIT Request phase

- A COORDINATOR send a QUERY TO COMMIT to each PARTICIPANT of the transaction, and waits for all replies.
- Each PARTICIPANT:
  - performs the transaction up to the point in which he's asked to COMMIT.
  - replies with a message:
    - AGREEMENT (if he would COMMIT the transaction).
    - ABORT (if he would ROLLBACK the transaction).

Two Phase Commit: Execution

Phase 2: COMMIT / ROLLBACK phase

- The COORDINATOR decides wheter to COMMIT or ROLLBACK the transaction, and issues the outcome to each PARTICIPANT.
- Each PARTICIPANT:
  - follow the issued COMMIT or ROLLBACK, releasing lock and resources.
  - sends an ACKNOWLEDGEMENT back to the COORDINATOR.
- The COORDINATOR completes the transaction when all the ACKNOWLEDGEMENTS have been received.

Two Phase Commit: Execution

Phase 2: COMMIT / ROLLBACK phase

- The COORDINATOR decides wheter to COMMIT or ROLLBACK the transaction, and issues the outcome to each PARTICIPANT.
- Each PARTICIPANT:
  - follow the issued COMMIT or ROLLBACK, releasing lock and resources.
  - sends an ACKNOWLEDGEMENT back to the COORDINATOR.
- The COORDINATOR completes the transaction when all the ACKNOWLEDGEMENTS have been received.

Two Phase Commit: Execution

Phase 2: COMMIT / ROLLBACK phase

- The COORDINATOR decides wheter to COMMIT or ROLLBACK the transaction, and issues the outcome to each PARTICIPANT.
- Each PARTICIPANT:
  - follow the issued COMMIT or ROLLBACK, releasing lock and resources.
  - sends an ACKNOWLEDGEMENT back to the COORDINATOR.
- The COORDINATOR completes the transaction when all the ACKNOWLEDGEMENTS have been received.

Two Phase Commit: Analysis

How much does Consistency cost?

Downsides

- It **blocks**!
- The COORDINATOR is a **Single Point of Failure**.
- Horizontal scaling isn't so effective.

Solutions in RDBMS world

- 2PC optimizations: Recursive 2PC, Dynamic 2PC, .. :
  - work better, but they are just optimizations.
- Three Phase Commit and optimizations:
  - more resilient to failures, but still block.

Two Phase Commit: Analysis

How much does Consistency cost?

Downsides

- It **blocks**!
- The COORDINATOR is a **Single Point of Failure**.
- Horizontal scaling isn't so effective.

Solutions in RDBMS world

- 2PC optimizations: Recursive 2PC, Dynamic 2PC, .. :
  - work better, but they are just optimizations.
- Three Phase Commit and optimizations:
  - more resilient to failures, but still block.

Two Phase Commit: Analysis

How much does Consistency cost?

Downsides

- It **blocks**!
- The COORDINATOR is a **Single Point of Failure**.
- Horizontal scaling isn't so effective.

Solutions in RDBMS world

- 2PC optimizations: Recursive 2PC, Dynamic 2PC, .. :
  - work better, but they are just optimizations.
- Three Phase Commit and optimizations:
  - more resilient to failures, but still block.

Two Phase Commit: Analysis

How much does Consistency cost?

Downsides

- It **blocks**!
- The COORDINATOR is a **Single Point of Failure**.
- Horizontal scaling isn't so effective.

Solutions in RDBMS world

- 2PC optimizations: Recursive 2PC, Dynamic 2PC, .. :
  - work better, but they are just optimizations.
- Three Phase Commit and optimizations:
  - more resilient to failures, but still block.

Two Phase Commit: Analysis

How much does Consistency cost?

Downsides

- It **blocks**!
- The COORDINATOR is a **Single Point of Failure**.
- Horizontal scaling isn't so effective.

Solutions in RDBMS world

- 2PC optimizations: Recursive 2PC, Dynamic 2PC, .. :
  - work better, but they are just optimizations.
- Three Phase Commit and optimizations:
  - more resilient to failures, but still block.

or ...

Maybe ACID is too much for some scenarios.

Introducing Cassandra

Cassandra facts

Timeline

2007 Started at Facebook, to solve performance and scalability issues in the company's inbox search.

2008 Open sourced, commit rights to Facebook staff.

2009 Apache Incubator project.

2010 Apache Top-level project.

Feedbacks from the trenches

Used by Facebook, Twitter, Digg, Reddit, Rackspace, Cisco, ...

Largest cluster: $>$ 300TB, 400 nodes.

Cassandra facts

Timeline

2007 Started at Facebook, to solve performance and scalability issues in the company's inbox search.

2008 Open sourced, commit rights to Facebook staff.

2009 Apache Incubator project.

2010 Apache Top-level project.

Feedbacks from the trenches

Used by Facebook, Twitter, Digg, Reddit, Rackspace, Cisco, ...

Largest cluster: $>$ 300TB, 400 nodes.

Cassandra facts

Timeline

2007 Started at Facebook, to solve performance and scalability issues in the company's inbox search.

2008 Open sourced, commit rights to Facebook staff.

2009 Apache Incubator project.

2010 Apache Top-level project.

Feedbacks from the trenches

Used by Facebook, Twitter, Digg, Reddit, Rackspace, Cisco, ...

Largest cluster: $>$ 300TB, 400 nodes.

Cassandra facts

Timeline

2007 Started at Facebook, to solve performance and scalability issues in the company's inbox search.

2008 Open sourced, commit rights to Facebook staff.

2009 Apache Incubator project.

2010 Apache Top-level project.

Feedbacks from the trenches

Used by Facebook, Twitter, Digg, Reddit, Rackspace, Cisco, ...

Largest cluster: $>$ 300TB, 400 nodes.

Cassandra facts

Timeline

2007 Started at Facebook, to solve performance and scalability issues in the company's inbox search.

2008 Open sourced, commit rights to Facebook staff.

2009 Apache Incubator project.

2010 Apache Top-level project.

Feedbacks from the trenches

Used by Facebook, Twitter, Digg, Reddit, Rackspace, Cisco, ...

Largest cluster: $>$ 300TB, 400 nodes.

- Data model based on Google's **Bigtable**.

- Distribution model based on Amazon's **Dynamo**.

Key Properties

NoSQL Data Model essentially a multidimensional hash-map.

Decentralized nodes are peers, each node can satisfy any request, **no single-point of failure**.

Elastic Scalability **horizontally scalable**, both **up and down**, almost linearly.

Fault Tolerance data replication, node add/replacement with no downtime.

High Availability

Tuneable Consistency:

- by cluster configuration (e.g. replication factor);
- by client, per-operation (e.g consistency level).

(see more later about tuneable consistency)

Key Properties

NoSQL Data Model essentially a multidimensional hash-map.

Decentralized nodes are peers, each node can satisfy any request, **no single-point of failure**.

Elastic Scalability **horizontally scalable**, both **up and down**, almost linearly.

Fault Tolerance data replication, node add/replacement with no downtime.

High Availability

Tuneable Consistency:

- by cluster configuration (e.g. replication factor);
- by client, per-operation (e.g consistency level).

(see more later about tuneable consistency)

Key Properties

NoSQL Data Model essentially a multidimensional hash-map.

Decentralized nodes are peers, each node can satisfy any request, **no single-point of failure**.

Elastic Scalability **horizontally scalable**, both **up and down**, almost linearly.

Fault Tolerance data replication, node add/replacement with no downtime.

High Availability

Tuneable Consistency:

- by cluster configuration (e.g. replication factor);
- by client, per-operation (e.g consistency level).

(see more later about tuneable consistency)

Key Properties

NoSQL Data Model essentially a multidimensional hash-map.

Decentralized nodes are peers, each node can satisfy any request, **no single-point of failure**.

Elastic Scalability **horizontally scalable**, both **up and down**, almost linearly.

Fault Tolerance data replication, node add/replacement with no downtime.

High Availability

Tuneable Consistency:

- by cluster configuration (e.g. replication factor);
- by client, per-operation (e.g consistency level).

(see more later about tuneable consistency)

Key Properties

NoSQL Data Model essentially a multidimensional hash-map.

Decentralized nodes are peers, each node can satisfy any request, **no single-point of failure**.

Elastic Scalability **horizontally scalable**, both **up and down**, almost linearly.

Fault Tolerance data replication, node add/replacement with no downtime.

High Availability

Tuneable Consistency:

- by cluster configuration (e.g. replication factor);
- by client, per-operation (e.g consistency level).

(see more later about tuneable consistency)

Key Properties

NoSQL Data Model essentially a multidimensional hash-map.

Decentralized nodes are peers, each node can satisfy any request, **no single-point of failure**.

Elastic Scalability **horizontally scalable**, both **up and down**, almost linearly.

Fault Tolerance data replication, node add/replacement with no downtime.

High Availability

Tuneable Consistency:

- by cluster configuration (e.g. replication factor);
- by client, per-operation (e.g consistency level).

(see more later about tuneable consistency)

Outline

Data model

... in less than 5 minutes

Keyspace Outermost container for data *(similar to schema in RDB world)*.

Column Family Collection of columns *(similar to table in RDB world)*.

Column Map from row keys to values. Value updates are timestamped. Timestamps are provided by clients.

SuperColumn Special kind of columns which hold normal columns. *Not discussed here*.

Data model

... in less than 5 minutes

Keyspace Outermost container for data *(similar to schema in RDB world)*.

Column Family Collection of columns *(similar to table in RDB world)*.

Column Map from row keys to values. Value updates are timestamped. Timestamps are provided by clients.

SuperColumn Special kind of columns which hold normal columns. *Not discussed here*.

Data model

... in less than 5 minutes

Keyspace Outermost container for data *(similar to schema in RDB world)*.

Column Family Collection of columns *(similar to table in RDB world)*.

Column Map from row keys to values. Value updates are timestamped. Timestamps are provided by clients.

SuperColumn Special kind of columns which hold normal columns. *Not discussed here.*

Data model

... in less than 5 minutes

Keyspace Outermost container for data *(similar to schema in RDB world)*.

Column Family Collection of columns *(similar to table in RDB world)*.

Column Map from row keys to values. Value updates are timestamped. Timestamps are provided by clients.

SuperColumn Special kind of columns which hold normal columns. *Not discussed here*.

Data model, JSON-ified

```
cloud_app: {                                # Keyspace       (KS)
  User: {                                   # Column Family  (CF)
    foo: {                                  # Row key        (RK)
      mail:     "foo@example.com" # Column name->value
    }
    bar: {                                  # Another row
      mail:     "bar@example.com" # Different columns
      locale:   "it_IT"           #   are allowed in
      timezone: "GMT+1"           #   the same CF.
    }
  }
  Server: {                                 # Another CF
    srv0: { ... }, srv1: { ... }, ...
  }
}
```

Data model

... in less than 5 minutes

Notes

- Different rows can have different columns in the same CF *(schema-less "tables")*.
- Data growth is often in the "column direction" *(vs. row in RDBs)*. In Cassandra, **lots** of columns mean $\sim 10^8$ columns. Limit is $2 \cdot 10^9$.
- Columns are sorted by their name *(efficient column-slice queries, heavily exploited in data design)*.

Data model

... in less than 5 minutes

Notes

- Different rows can have different columns in the same CF *(schema-less "tables")*.
- Data growth is often in the "column direction" *(vs. row in RDBs)*. In Cassandra, **lots** of columns mean $\sim 10^8$ columns. Limit is $2 \cdot 10^9$.
- Columns are sorted by their name *(efficient column-slice queries, heavily exploited in data design)*.

Data model

... in less than 5 minutes

Notes

- Different rows can have different columns in the same CF *(schema-less "tables")*.
- Data growth is often in the "column direction" *(vs. row in RDBs)*. In Cassandra, **lots** of columns mean $\sim 10^8$ columns. Limit is $2 \cdot 10^9$.
- Columns are sorted by their name *(efficient column-slice queries, heavily exploited in data design)*.

Outline

Data space

The data space

- Is represented as a ring of (2^{127}) tokens. Each token corresponds to a row key.
- Is sliced into ranges, with number of ranges = number of nodes.
- Each node covers a slice of the token space.
- A component called **Partitioner** maps row keys to tokens, affecting data distribution among the nodes.

Data space

The data space

- Is represented as a ring of (2^{127}) tokens. Each token corresponds to a row key.
- Is sliced into ranges, with number of ranges = number of nodes.
- Each node covers a slice of the token space.
- A component called **Partitioner** maps row keys to tokens, affecting data distribution among the nodes.

Data space

The data space

- Is represented as a ring of (2^{127}) tokens. Each token corresponds to a row key.
- Is sliced into ranges, with number of ranges = number of nodes.

- Each node covers a slice of the token space.
- A component called **Partitioner** maps row keys to tokens, affecting data distribution among the nodes.

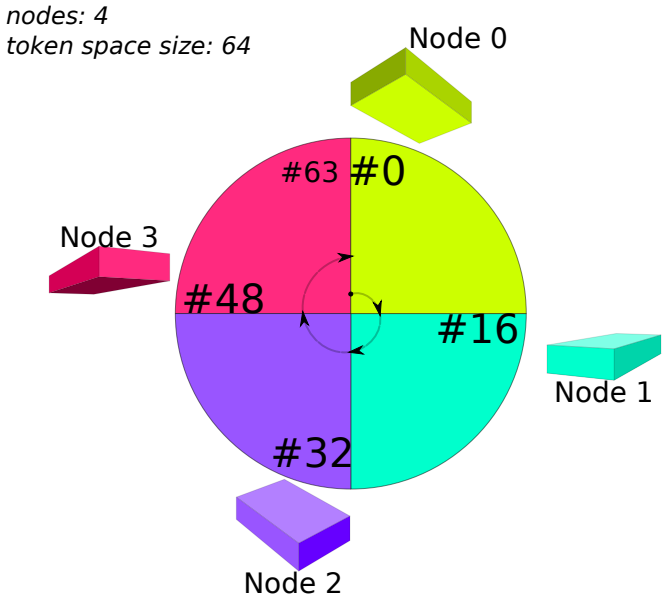
Data space

The data space

- Is represented as a ring of (2^{127}) tokens. Each token corresponds to a row key.
- Is sliced into ranges, with number of ranges = number of nodes.

- Each node covers a slice of the token space.
- A component called **Partitioner** maps row keys to tokens, affecting data distribution among the nodes.

Data space
The ring



Data distribution

- **Row oriented Sharding and Replication** of data.
- Main knobs:
  - cluster-wide configuration:

    Partitioner controls how data is partitioned *(which node belongs row key #k?)*
  - per-keyspace configuration:

    Replication Factor controls how many replicas are stored.
    Replication Strategy controls in which nodes replicated data goes.

Data distribution

- **Row oriented Sharding and Replication** of data.
- Main knobs:
  - cluster-wide configuration:

    Partitioner controls how data is partitioned *(which node belongs row key #k?)*
  - per-keyspace configuration:

    Replication Factor controls how many replicas are stored.
    Replication Strategy controls in which nodes replicated data goes.

Data partitioning
Partitioner

`org.apache.cassandra.dht.IPartitioner`

Two main kinds:

- `RandomPartitioner`: key -> **MD5 hash** -> token.

  Pros: Keys are evenly spread.
  Cons: Key-range queries may access nodes in very different locations, results are unsorted.

- `OrderPreservingPartitioner`: key -> **utf-8 string** -> token.

  Pros: Query results are sorted by key.
  Cons: Unevenly spread data among nodes, needs periodic rebalancing.

Data partitioning
Partitioner

`org.apache.cassandra.dht.IPartitioner`

Two main kinds:

- `RandomPartitioner`: key -> **MD5 hash** -> token.

  Pros: Keys are evenly spread.
  Cons: Key-range queries may access nodes in very different locations, results are unsorted.

- `OrderPreservingPartitioner`: key -> **utf-8 string** -> token.

  Pros: Query results are sorted by key.
  Cons: Unevenly spread data among nodes, needs periodic rebalancing.

Data partitioning
Partitioner

`org.apache.cassandra.dht.IPartitioner`

Two main kinds:

- `RandomPartitioner`: key -> **MD5 hash** -> token.

  Pros: Keys are evenly spread.
  Cons: Key-range queries may access nodes in very different locations, results are unsorted.

- `OrderPreservingPartitioner`: key -> **utf-8 string** -> token.

  Pros: Query results are sorted by key.
  Cons: Unevenly spread data among nodes, needs periodic rebalancing.

Data replication
Replication Strategy

`org.apache.cassandra.locator.AbstractReplicationStrategy`

Strategy GoF pattern, some implementations:

- `SimpleStrategy`: network unaware, follow node numbering.
- `OldNetworkTopologyStrategy`: datacenter-aware, first replica goes in a different datacenter.
- `NetworkTopologyStrategy` *(Cassandra 0.7+)*: datacenter-aware, allows to specify how many replicas go into each datacenter.

Data replication
Replication Strategy

`org.apache.cassandra.locator.AbstractReplicationStrategy`

Strategy GoF pattern, some implementations:

- `SimpleStrategy`: network unaware, follow node numbering.
- `OldNetworkTopologyStrategy`: datacenter-aware, first replica goes in a different datacenter.
- `NetworkTopologyStrategy` *(Cassandra 0.7+)*: datacenter-aware, allows to specify how many replicas go into each datacenter.

Data replication
Replication Strategy

`org.apache.cassandra.locator.AbstractReplicationStrategy`

Strategy GoF pattern, some implementations:

- `SimpleStrategy`: network unaware, follow node numbering.
- `OldNetworkTopologyStrategy`: datacenter-aware, first replica goes in a different datacenter.
- `NetworkTopologyStrategy` *(Cassandra 0.7+)*: datacenter-aware, allows to specify how many replicas go into each datacenter.

Data replication
Replication Strategy

`org.apache.cassandra.locator.AbstractReplicationStrategy`

Strategy GoF pattern, some implementations:

- `SimpleStrategy`: network unaware, follow node numbering.
- `OldNetworkTopologyStrategy`: datacenter-aware, first replica goes in a different datacenter.
- `NetworkTopologyStrategy` *(Cassandra 0.7+)*: datacenter-aware, allows to specify how many replicas go into each datacenter.

Outline

Request

- A client connects to a node, and performs a request.
- The node assumes the role of COORDINATOR for the operation.
- The node acts to fullfill the request.
- Communication to other nodes is generally involved.

Tuneable Consistency

Tuneable tradeoff between Consistency, Availability, Latency
(remember $P?(A|C : L|C)$?)

Consistency Level

Each client request has an associated Consistency Level: the minimum number of replicas that should respond.

CL	Read	Write
ZERO	☹	No Wait, No Guarantees.
ANY	☹	First response, including Hinted Handoff *(discussed later)*.
ONE	First response.	First response.
QUORUM	$RF/2 + 1$ replicas.	$RF/2 + 1$ replicas.
ALL	All replicas. Fail-fast.	All replicas. Fail-fast.

CL = Consistency Level.

RF = Replication Factor.

Write

- CL to `ALL` or `QUORUM` guarantee writes to a sufficient number of replicas.
- `QUORUM` statistically lowers latency and is more fault tolerant then `ALL`.
- Cassandra guarantees atomic writes on a single node, row key basis.
- **Failed writes don't cause roll-backs** in any of the eventual nodes where the write succeeded.

Write: Hinted Handoff

Happens when:

- The COORDINATOR detects that a replica node for the write operation is down,
- **and** the write operation is set to CL: ANY or higher.

How it works

1. The COORDINATOR writes down an ***hint*** for the operation.
1. The hint will be ***handoff*** to the target node once it will become available again.

Write: Hinted Handoff

Happens when:

- The COORDINATOR detects that a replica node for the write operation is down,
- **and** the write operation is set to CL: ANY or higher.

How it works

1. The COORDINATOR writes down an ***hint*** for the operation.
1. The hint will be ***handoff*** to the target node once it will become available again.

Write: Hinted Handoff

Happens when:

- The COORDINATOR detects that a replica node for the write operation is down,
- **and** the write operation is set to CL: ANY or higher.

How it works

1. The COORDINATOR writes down an ***hint*** for the operation.
1. The hint will be ***handoff*** to the target node once it will become available again.

Write: Hinted Handoff

Happens when:

- The COORDINATOR detects that a replica node for the write operation is down,
- **and** the write operation is set to CL: ANY or higher.

How it works

1. The COORDINATOR writes down an ***hint*** for the operation.
1. The hint will be ***handoff*** to the target node once it will become available again.

Write: Hinted Handoff

Why?

- Reduce the time required for a temporarily failed node to become consistent again.
- Provide **extreme write Availability** when Consistency is not required *(CL ANY)*.

Issue

- When a failed node becomes available again, could be flooded by hints from other nodes.
- It's possible to disable HH or reduce its priority of over "normal" writes.

Write: Hinted Handoff

Why?

- Reduce the time required for a temporarily failed node to become consistent again.
- Provide **extreme write Availability** when Consistency is not required *(CL ANY)*.

Issue

- When a failed node becomes available again, could be flooded by hints from other nodes.
- It's possible to disable HH or reduce its priority of over "normal" writes.

Write: Hinted Handoff

Why?

- Reduce the time required for a temporarily failed node to become consistent again.
- Provide **extreme write Availability** when Consistency is not required *(CL ANY)*.

Issue

- When a failed node becomes available again, could be flooded by hints from other nodes.
- It's possible to disable HH or reduce its priority of over "normal" writes.

Write: Hinted Handoff

Why?

- Reduce the time required for a temporarily failed node to become consistent again.
- Provide **extreme write Availability** when Consistency is not required *(CL ANY)*.

Issue

- When a failed node becomes available again, could be flooded by hints from other nodes.
- It's possible to disable HH or reduce its priority of over "normal" writes.

Read

The COORDINATOR contacts only a number of replicas specified by the Consistency Level.

If the number of replicas involved is:

- 1: a **background thread** performs **Read Repair** to ensure Consistency with other replicas.
- > 1:
  - if the involved replicas are inconsistent, a **foreground** *(blocking)* **Read Repair** is performed.
  - the most recent value is returned.
  - a **background thread** performs **Read Repair** between all replicas.

Read

The COORDINATOR contacts only a number of replicas specified by the Consistency Level.
If the number of replicas involved is:

- 1: a **background thread** performs **Read Repair** to ensure Consistency with other replicas.
- > 1:
  - if the involved replicas are inconsistent, a **foreground** *(blocking)* **Read Repair** is performed.
  - the most recent value is returned.
  - a **background thread** performs **Read Repair** between all replicas.

Read

The COORDINATOR contacts only a number of replicas specified by the Consistency Level.
If the number of replicas involved is:

- 1: a **background thread** performs **Read Repair** to ensure Consistency with other replicas.
- > 1:
  - if the involved replicas are inconsistent, a **foreground** *(blocking)* **Read Repair** is performed.
  - the most recent value is returned.
  - a **background thread** performs **Read Repair** between all replicas.

Read

The COORDINATOR contacts only a number of replicas specified by the Consistency Level.
If the number of replicas involved is:

- 1: a **background thread** performs **Read Repair** to ensure Consistency with other replicas.
- > 1:
  - if the involved replicas are inconsistent, a **foreground** *(blocking)* **Read Repair** is performed.
  - the most recent value is returned.
  - a **background thread** performs **Read Repair** between all replicas.

Read

The COORDINATOR contacts only a number of replicas specified by the Consistency Level.
If the number of replicas involved is:

- 1: a **background thread** performs **Read Repair** to ensure Consistency with other replicas.
- > 1:
  - if the involved replicas are inconsistent, a **foreground** *(blocking)* **Read Repair** is performed.
  - the most recent value is returned.
  - a **background thread** performs **Read Repair** between all replicas.

Read Repair

It's a **timestamp-based** way to achieve **replica conciliation**.

Main issue

- As said before, timestamps are provided by clients.
- Clients are part of the system: they **must** have **low clock drifts**. *(needs good design of `NTP` network)*

Future improvements

Dynamo has **vector clocks**.
Next Cassandra versions may adopt **vector clocks** (or **version vectors**) as well.

Read Repair

It's a **timestamp-based** way to achieve **replica conciliation**.

Main issue

- As said before, timestamps are provided by clients.
- Clients are part of the system: they **must** have **low clock drifts**. *(needs good design of NTP network)*

Future improvements

Dynamo has **vector clocks**.
Next Cassandra versions may adopt **vector clocks** (or **version vectors**) as well.

Read Repair

It's a **timestamp-based** way to achieve **replica conciliation**.

Main issue

- As said before, timestamps are provided by clients.
- Clients are part of the system: they **must** have **low clock drifts**. *(needs good design of `NTP` network)*

Future improvements

Dynamo has **vector clocks**.
Next Cassandra versions may adopt **vector clocks** (or **version vectors**) as well.

Read Repair

It's a **timestamp-based** way to achieve **replica conciliation**.

Main issue

- As said before, timestamps are provided by clients.
- Clients are part of the system: they **must** have **low clock drifts**. *(needs good design of `NTP` network)*

Future improvements

Dynamo has **vector clocks**.
Next Cassandra versions may adopt **vector clocks** (or **version vectors**) as well.

Where to put latency?

Cassandra choose low latency writes over low latency reads *(Read Repair over Write Repair)*.

Cache write-back model *(Borrowed from BigTable)*:

1. Writes are written to a commit log,
2. then written to in-memory tables (memtables),
3. then flushed in persistent storage to read-only tables (SSTables, sorted strings tables).

Periodic compaction of SSTables.

Where to put latency?

Cassandra choose low latency writes over low latency reads *(Read Repair over Write Repair)*.

Cache write-back model *(Borrowed from BigTable)*:

1. Writes are written to a commit log,
2. then written to in-memory tables (memtables),
3. then flushed in persistent storage to read-only tables (SSTables, sorted strings tables).

Periodic compaction of SSTables.

Consistency

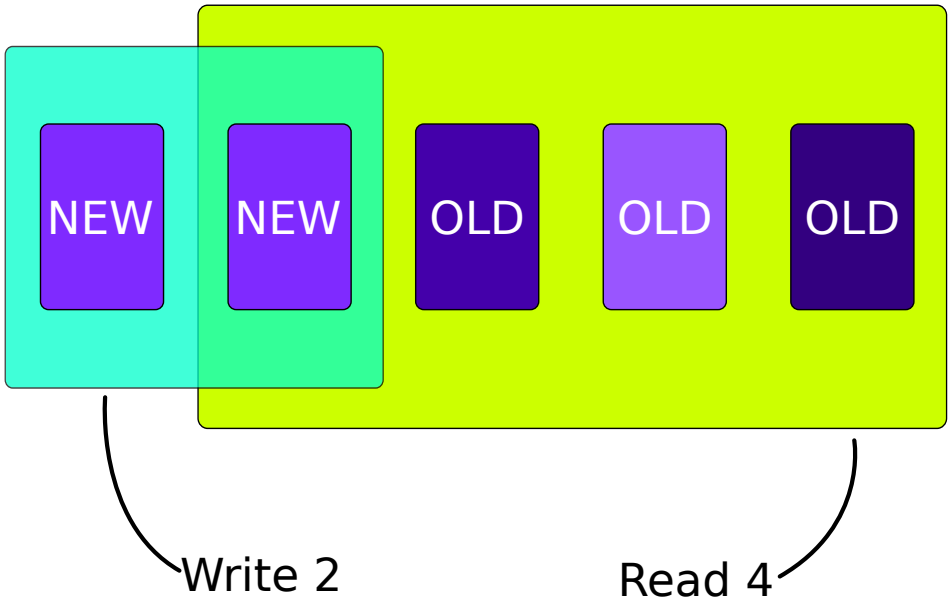
- $W + R \leq RF$: **Weak consistency**: clients may read stale values.
- $W + R > RF$: **Strong consistency**: clients will always read the most recent write.

Consistency

- $W + R \leq RF$: **Weak consistency**: clients may read stale values.
- $W + R > RF$: **Strong consistency**: clients will always read the most recent write.

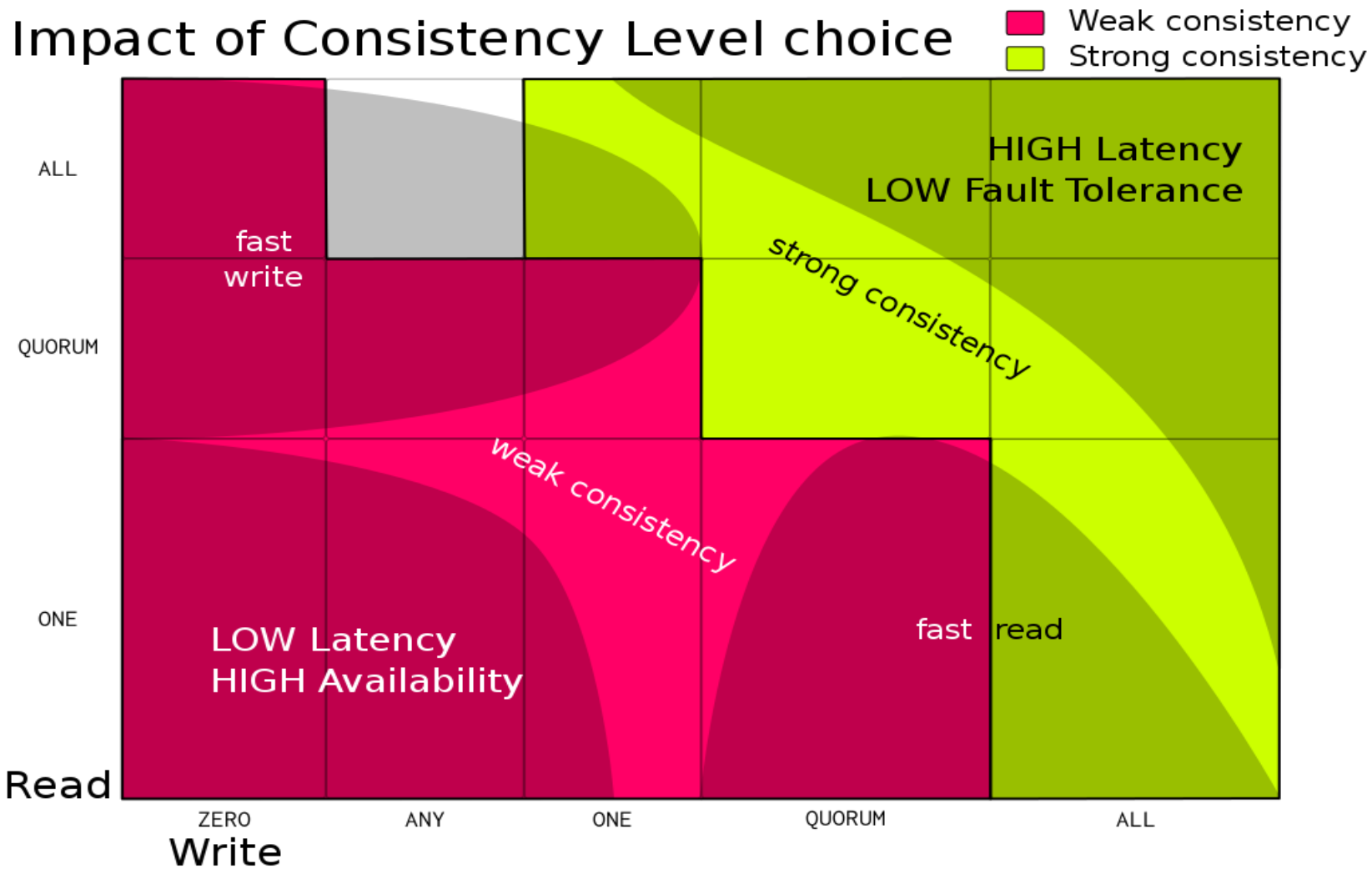
Consistency
Strong consistency example

Replication factor: 5



Read includes at least 1 up-to-date value

Tuning the Consistency Level



Outline

Gossip

Intra-ring epidemic protocol.

Why?

- **Robust** way to spread nodes' statuses among the cluster.

- Runs on a configurable time interval (default: every 1 second). **Negligible network traffic.**

- Peer-to-peer
  - 1 to 3 peers contacted on every session by default,
  - **random component on peer choice** + tricks for partition tolerance.

Good description on: Gossip Architecture on Apache's Cassandra Wiki

Gossip

Intra-ring epidemic protocol.

Why?

- **Robust** way to spread nodes' statuses among the cluster.

- Runs on a configurable time interval (default: every 1 second). **Negligible network traffic.**

- Peer-to-peer
  - 1 to 3 peers contacted on every session by default,
  - **random component on peer choice** + tricks for partition tolerance.

Good description on: Gossip Architecture on Apache's Cassandra Wiki

Gossip

Intra-ring epidemic protocol.

Why?

- **Robust** way to spread nodes' statuses among the cluster.

- Runs on a configurable time interval (default: every 1 second). **Negligible network traffic.**

- Peer-to-peer
  - 1 to 3 peers contacted on every session by default,
  - **random component on peer choice** + tricks for partition tolerance.

Good description on: Gossip Architecture on Apache's Cassandra Wiki

Gossip

Intra-ring epidemic protocol.

Why?

- **Robust** way to spread nodes' statuses among the cluster.

- Runs on a configurable time interval (default: every 1 second). **Negligible network traffic.**

- Peer-to-peer
  - 1 to 3 peers contacted on every session by default,
  - **random component on peer choice** + tricks for partition tolerance.

Good description on: Gossip Architecture on Apache's Cassandra Wiki

Versioned state tracking:

- `Generation Number` (`GN`): is incremented on every node boot.
- `Version Number` (`VN`): is reset on node boot.

Example of state information for node 10.0.0.1

```
EndPointState 10.0.0.1
  HeartBeatState: generation 1259909635, version 325
  ApplicationState "load-information": 5.2,
      generation 1259909635, version 45
  ApplicationState "bootstrapping": bxLpassF3XD8Kyks,
      generation 1259909635, version 56
  ApplicationState "normal": bxLpassF3XD8Kyks,
      generation 1259909635, version 87
```

Every "state descriptor" is versioned with `GN` and `VN`.

Versioned state tracking:

- `Generation Number` (`GN`): is incremented on every node boot.
- `Version Number` (`VN`): is reset on node boot.

Example of state information for node 10.0.0.1

```
EndPointState 10.0.0.1
  HeartBeatState: generation 1259909635, version 325
  ApplicationState "load-information": 5.2,
      generation 1259909635, version 45
  ApplicationState "bootstrapping": bxLpassF3XD8Kyks,
      generation 1259909635, version 56
  ApplicationState "normal": bxLpassF3XD8Kyks,
      generation 1259909635, version 87
```

Every "state descriptor" is versioned with `GN` and `VN`.

Gossip
Gossip Digest

A node $\mathcal{N}$'s **Gossip Digest** for target node $\mathcal{T}$ reflects the current knowledge that $\mathcal{N}$ has about $\mathcal{T}$.

A digest about $\mathcal{T}$ includes:

- $\mathcal{T}$'s address,
- The greatest (latest) `GN` and `VN` that $\mathcal{N}$ has seen about $\mathcal{T}$.

Gossip Session: 1/3

Node $\mathcal{I}$ is the GOSSIP INITIATOR, Node $\mathcal{R}$ is a GOSSIP RESPONDER.

$\mathcal{I}$: Send `GossipDigestSynMessage` to $\mathcal{R}$

Includes Gossip Digests about a group of nodes (including $\mathcal{I}$ and $\mathcal{R}$).

Gossip Session: 1/3

Node $\mathcal{I}$ is the GOSSIP INITIATOR, Node $\mathcal{R}$ is a GOSSIP RESPONDER.

$\mathcal{I}$: Send `GossipDigestSynMessage` to $\mathcal{R}$

Includes Gossip Digests about a group of nodes (including $\mathcal{I}$ and $\mathcal{R}$).

Gossip Session: 2/3

Examining the incoming Syn message, $\mathcal{R}$ determines:

- what status updates are needed by $\mathcal{I}$.
- what status updates can be required from $\mathcal{I}$.

$\mathcal{R}$: Reply to $\mathcal{I}$ with `GossipDigestAckMessage`

Includes:

- Gossip Digests to require status updates,
- Status updates.

Gossip Session: 3/3

$\mathcal{I}$: Reply to $\mathcal{R}$ with `GossipDigestAck2Message`

Includes status updates required by $\mathcal{R}$.

φ-accrual Failure detection

Hayashibara, Défago, Yared, Katayama

Heartbeat-based adaptive failure detector.

Rationale

- Factor out the failure detector as a *component* / *service*.
- Bring *uncertainty*, *adaptivity*, *flexibility* in failure detection: boost **fault tolerance**.
- Move failure status from the boolean domain to a **continuous domain**.

φ-accrual Failure detection

Hayashibara, Défago, Yared, Katayama

Heartbeat-based adaptive failure detector.

Rationale

- Factor out the failure detector as a *component* / *service*.
- Bring *uncertainty*, *adaptivity*, *flexibility* in failure detection: boost **fault tolerance**.
- Move failure status from the boolean domain to a **continuous domain**.

φ-accrual Failure detection

Hayashibara, Défago, Yared, Katayama

Heartbeat-based adaptive failure detector.

Rationale

- Factor out the failure detector as a *component* / *service*.
- Bring *uncertainty*, *adaptivity*, *flexibility* in failure detection: boost **fault tolerance**.
- Move failure status from the boolean domain to a **continuous domain**.

φ-accrual Failure detection

Hayashibara, Défago, Yared, Katayama

Heartbeat-based adaptive failure detector.

Rationale

- Factor out the failure detector as a *component* / *service*.
- Bring *uncertainty*, *adaptivity*, *flexibility* in failure detection: boost **fault tolerance**.
- Move failure status from the boolean domain to a **continuous domain**.

φ-accrual Failure detection

- The value emitted by the failure detector is the **degree of confidence** that the monitored process has crashed.
- Clients of the FD service can interpret the value as they wish.
- Conservative vs. aggressive *(analogy with control systems: precise vs. fast)*? *It's up to the clients*.

Why accrual?

Accrual *(noun)* The act or process of accumulating.
Synonim: ***accumulation***.

If the target process fails, the φ value *is guaranteed to increase over time*.

φ-accrual Failure detection

- The value emitted by the failure detector is the **degree of confidence** that the monitored process has crashed.
- Clients of the FD service can interpret the value as they wish.
- Conservative vs. aggressive *(analogy with control systems: precise vs. fast)*? *It's up to the clients.*

Why accrual?

Accrual *(noun)* The act or process of accumulating.
Synonim: ***accumulation***.

If the target process fails, the φ value *is guaranteed to increase over time.*

φ-accrual Failure detection

- The value emitted by the failure detector is the **degree of confidence** that the monitored process has crashed.
- Clients of the FD service can interpret the value as they wish.
- Conservative vs. aggressive *(analogy with control systems: precise vs. fast)*? *It's up to the clients*.

Why accrual?

Accrual *(noun)* The act or process of accumulating.
Synonim: ***accumulation***.

If the target process fails, the φ value *is guaranteed to increase over time*.

φ-accrual Failure detection

- The value emitted by the failure detector is the **degree of confidence** that the monitored process has crashed.
- Clients of the FD service can interpret the value as they wish.
- Conservative vs. aggressive *(analogy with control systems: precise vs. fast)*? *It's up to the clients*.

Why accrual?

Accrual *(noun)* The act or process of accumulating.
Synonim: ***accumulation***.

If the target process fails, the φ value *is guaranteed to increase over time*.

Failure detection
Eventually perfect failure detection

A FD is said to be **Eventually Perfect** *(or to belong to class $\diamond P$)* if satisfies the following properties:

Class $\diamond P$ FD properties:

1. **Strong Completeness**: $\exists t_{susp}$ after which all failed processes are permanently suspected.
2. **Eventual Strong Accuracy**: $\exists t_{corr}$ after which all non-failed processes are not suspected.

The φ-accrual FD can be used to achieve $\diamond P$.

Failure detection
Eventually perfect failure detection

A FD is said to be **Eventually Perfect** *(or to belong to class $\diamond P$)* if satisfies the following properties:

Class $\diamond P$ FD properties:

1. **Strong Completeness**: $\exists t_{susp}$ after which all failed processes are permanently suspected.
2. **Eventual Strong Accuracy**: $\exists t_{corr}$ after which all non-failed processes are not suspected.

The φ-accrual FD can be used to achieve $\diamond P$.

Failure detection
Eventually perfect failure detection

A FD is said to be **Eventually Perfect** *(or to belong to class $\diamond P$)* if satisfies the following properties:

Class $\diamond P$ FD properties:

1. **Strong Completeness**: $\exists t_{susp}$ after which all failed processes are permanently suspected.
2. **Eventual Strong Accuracy**: $\exists t_{corr}$ after which all non-failed processes are not suspected.

The φ-accrual FD can be used to achieve $\diamond P$.

φ-accrual Failure detection
Working principles

- FD keeps a time window of last n inter-arrival times x_i, $i = 0..n$.
- Computes μ, σ^2 of the values currently in the window, subsuming a particular distribution X *(usually gaussian)*.
- φ value depends on the ***likelyhood that the next heartbeat will arrive***:

$$\varphi(t) = f(\mathcal{P}[X > (t - t_{last})])$$

φ-accrual Failure detection
Working principles

- FD keeps a time window of last n inter-arrival times x_i, $i = 0..n$.
- Computes μ, σ^2 of the values currently in the window, subsuming a particular distribution X *(usually gaussian)*.
- φ value depends on the ***likelyhood that the next heartbeat will arrive***:

$$\varphi(t) = f(\mathcal{P}[X > (t - t_{last})])$$

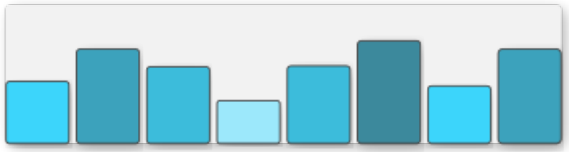
φ-accrual Failure detection
Working principles

- FD keeps a time window of last n inter-arrival times x_i, $i = 0..n$.
- Computes μ, σ^2 of the values currently in the window, subsuming a particular distribution X *(usually gaussian)*.
- φ value depends on the ***likelyhood that the next heartbeat will arrive***:

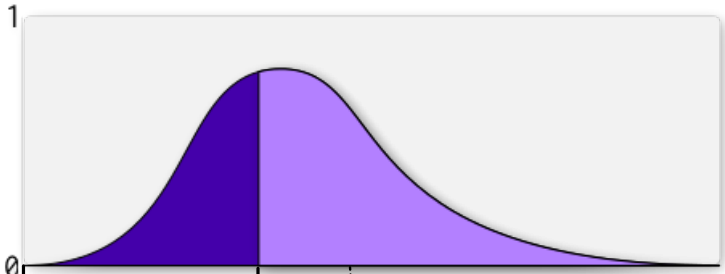
$$\varphi(t) = f\left(\mathcal{P}\left[X > (t - t_{last})\right]\right)$$

φ-accrual Failure detection

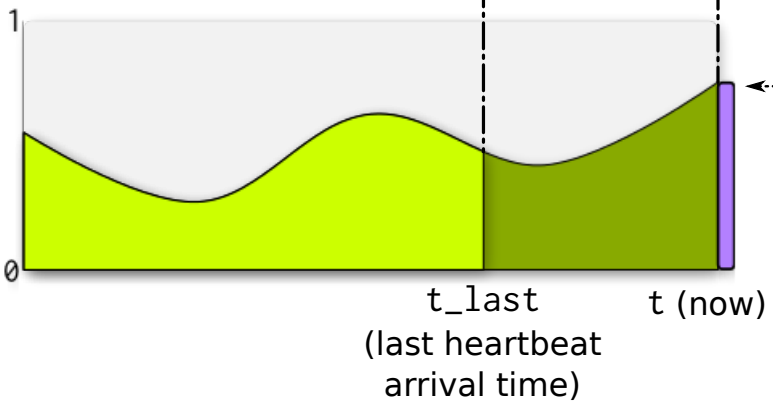
last n
heartbeat inter-arrival times



inter-arrival time
distribution



chance
of receiving
an heartbeat
over time



chance at t
that next heartbeat
will arrive

function φ(chance) → clients!

φ-accrual Failure detection

Clients can thresold the φ value from FD to achieve boolean failure detection (service is up / service is down).

In Cassandra ...

- The φ-thresold is configurable.
- The inter-arrival times are assumed to follow an **Exponential Distribution** *(maybe for its relation to a Poisson-ian process)*.

φ-accrual Failure detection

Clients can thresold the φ value from FD to achieve boolean failure detection (service is up / service is down).

In Cassandra ...

- The φ-thresold is configurable.
- The inter-arrival times are assumed to follow an **Exponential Distribution** *(maybe for its relation to a Poisson-ian process)*.

Outline

Anti-Entropy Service

Gossip-driven replica synchronization mechanism between neighboring nodes.

- Read-Repair alone isn't enough to keep replicas up-to-date.
- Triggered on major compactions *(when SSTables are merged)*, gossip-driven.
- Uses **Merkle Trees** to keep it fast.

Anti-Entropy Service

Gossip-driven replica synchronization mechanism between neighboring nodes.

- Read-Repair alone isn't enough to keep replicas up-to-date.
- Triggered on major compactions *(when SSTables are merged)*, gossip-driven.
- Uses **Merkle Trees** to keep it fast.

Merkle Trees

- A Merkle Tree is **a tree of hashes**:

  leaves are hashes of data blocks of a data structure.
  parent nodes are hashes of the leaves.

- Allows **fast comparison and location of differences** between two homogeneous data structures.

- Ideated by Ralph Merkle.
- *Used everywhere! BitCoin, GoogleWave, Gnutella(2), Riak, Shareaza, ..*

Merkle Trees

- A Merkle Tree is **a tree of hashes**:

  leaves are hashes of data blocks of a data structure.
  parent nodes are hashes of the leaves.

- Allows **fast comparison and location of differences** between two homogeneous data structures.

- Ideated by Ralph Merkle.
- *Used everywhere! BitCoin, GoogleWave, Gnutella(2), Riak, Shareaza, ..*

Merkle Trees

- A Merkle Tree is **a tree of hashes**:

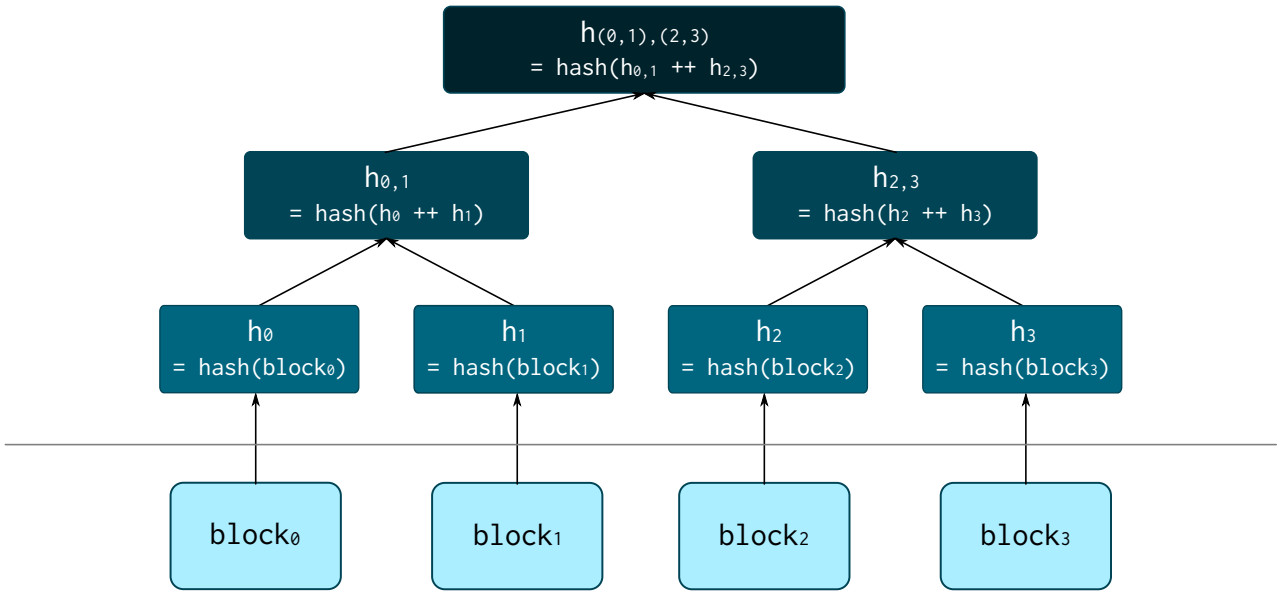
  leaves are hashes of data blocks of a data structure.
  parent nodes are hashes of the leaves.

- Allows **fast comparison and location of differences** between two homogeneous data structures.

- Ideated by Ralph Merkle.
- *Used everywhere! BitCoin, GoogleWave, Gnutella(2), Riak, Shareaza, ..*

Merkle Trees

Merkle Tree structure



Merkle Tree picture in Haskell

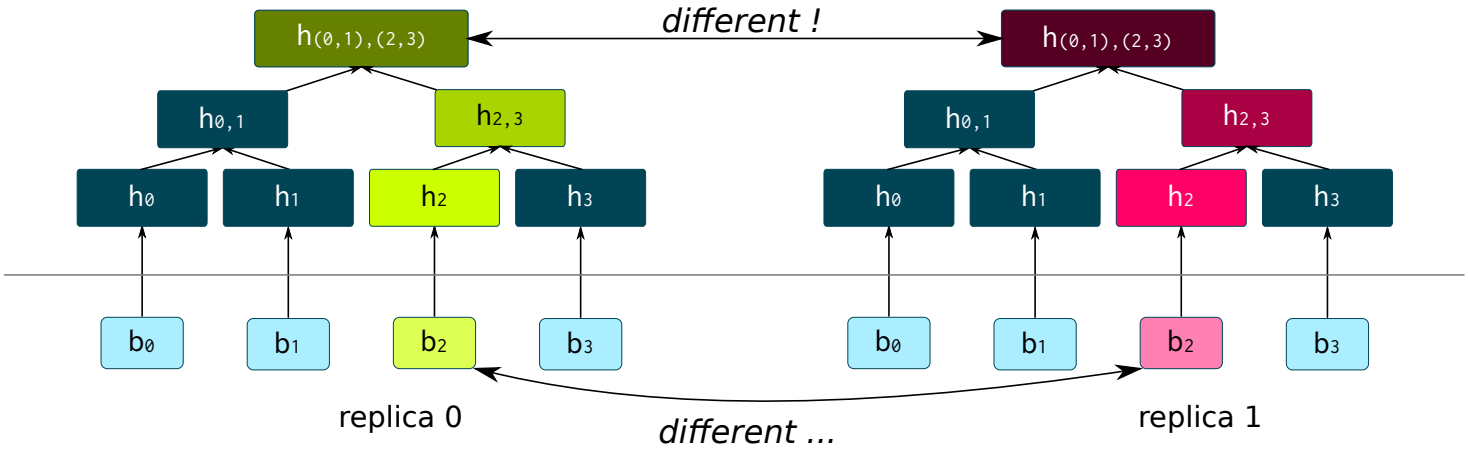
```
type Hash = ByteString
data BinTree a =   Leaf a
                 | Branch (BinTree a) a (BinTree a)
type MerkleTree = BinTree Hash

joinTrees :: MerkleTree -> MerkleTree -> MerkleTree
joinTrees left right =
    Branch left (hash (val left ++ val right)) right
```

A Merkle Tree detecting differences

Merkle Tree - based diff

differences in a data block <=> different hashes
differences "propagate" upwards



O(1) check for difference
O($\log_2(n)$) find the bad block

That's it!

Summary

- CAP theorem and thoughts about it.
- Cassandra in CAP context.
- Cassandra as an integrated system of engineering solutions.
- Something about Cassandra internals.

What next?

- More about Cassandra **internals** *(e.g. SEDA)*.
- Cassandra **rivals** in the fight for Big Data
  (e.g. Riak, Project Voldemort, ...).
- **Integration** in data processing environments *(e.g. Hadoop, ...)*.
- Distributed **transactions**
  (e.g. ZooKeeper, tuple-based coordination, ...).
- *Even more stuff* ☺

For Further Reading I

- Cassandra documentation from DataStax: datastax.com
- Apache's Cassandra Wiki: wiki.apache.org/cassandra/
- Cassandra Source Code: github.com/apache/cassandra
- D.Abadi's blog post on CAP Theorem

Cassandra: The Definitive Guide.

The φ Accrual Failure Detector.

Acknowledgements

- `#cassandra` IRC channel on Freenode
- ... and of course wikipedia.org

made with LyX and Inkscape on Gentoo Linux