Architecture and Implementation of Database Systems (Winter 2015/16)

Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Winter 2015/16

Part X

Distributed Databases

Distributed Databases

Parallel databases assume tight coupling between nodes.

→ *e.g.*, local cluster

→ main goal: parallel execution

Distributed databases have a slightly different motivation.

- geographically separate locations
- sites run full DBMS
- locality effects
- run local queries independently, but still allow for global queries
  → *e.g.*, for analytics
- increase availability / failure tolerance

Transparent Distribution

Want to keep distribution **transparent**:

- **Distributed Data Independence**
  - → Clients need not know how data is distributed or where objects are located.
  - → Automatic optimizer decides on **distributed query plans**.
- **Distributed Transaction Atomicity**
  - → Transactions across sites should be atomic.

Storing Data in a Distributed DBMS

Fragmentation:

- Break data into fragments and store them on sites.
  - → Exploit knowledge about data and access pattern

Replication:

- Place data/fragments on **multiple sites**
  - → increased **availability**
  - → **faster query evaluation**

Both are **trade-offs**:

- achievable **parallelism**; **communication** cost; **synchronization**; available **space**; **failure tolerance**

Horizontal Fragmentation

Each fragment consists of a **subset of rows** of the original relation.

Projects			
pid	Title	Office	Budget
1	Aquarius	London	16000
2	Eridanus	Paris	21000
3	Centaurus	Paris	17000
4	Andromeda	Rome	29000
5	Pegasus	London	23000

→

Projects			
pid	Title	Office	Budget
2	Eridanus	Paris	21000
3	Centaurus	Paris	17000
4	Andromeda	Rome	29000
1	Aquarius	London	16000
5	Pegasus	London	23000

Express each fragment as a **selection** on the input relation.

- $Projects_1 = \sigma_{Office=\text{`Paris'}}(Projects)$
- $Projects_2 = \sigma_{Office=\text{`Rome'}}(Projects)$
- $Projects_3 = \sigma_{Office=\text{`London'}}(Projects)$

Correctness Rules

Completeness:

- Each item in R can be found in (at least) one fragment R_i.

Reconstruction:

- It must be possible to re-construct R from the R_i.
  → "It must be possible to define a relational operator ∇ such that $R = \nabla(R_1, \dots, R_n)$."

Disjointness:

- Fragments do not overlap; *i.e.*, no data item is assigned to multiple fragments.

Horizontal Fragmentation

Horizontal fragmentation is defined by predicates p_i:

$$R_i = \sigma_{p_i}(R) \ .$$

How do we find predicates p_i such that the fragmentation is

- **correct**
- **well-suited** for the given application and data set?

Observation: Breaking a relation (fragment) into a **pair** of fragments ensures correctness:

$$R \quad \leadsto \quad R_1 = \sigma_p(R) \quad ; \quad R_2 = \sigma_{\neg p}(R) \ .$$

Horizontal Fragmentation

Idea: Derive p_i from **workload information**.

Step 1: Analyze workload

- **Qualitative Information:** Predicates used in queries
  - $\rightarrow$ Extract **simple predicates** of the form

$$s_j = \textit{attribute}\,\theta\,\textit{constant} \quad ,$$

    where $\theta \in \{=, <, \neq, \leq, >, \geq\}$.
  - $\rightarrow$ Observe that simple predicates are easy to negate.
  - $\rightarrow$ We refer to a conjunction of (negated) simple predicates as a **minterm**.
- **Quantitative Information:**
  - $\rightarrow$ **minterm selectivity**
  - $\rightarrow$ **access frequency** (of a minterm or a query)

Example

Queries:

Q_1:

```
SELECT Title
  FROM Projects
 WHERE Office = 'Paris'
```

Q_2:

```
SELECT Office
  FROM Projects
 WHERE Budget BETWEEN
       15000 AND 20000
```

Simple Predicates:

- $s_1 \equiv \mathit{Office} = \text{'Paris'}$
- $s_2 \equiv \mathit{Budget} \geq 15000$
- $s_3 \equiv \mathit{Budget} \leq 20000$

Horizontal Fragmentation: Enumerate Minterms

Step 2: Enumerate Possible Minterms

- Build all possible minterms with given simple predicates and their negation.

Example:

$$
\begin{aligned}
m_1 &\equiv \mathit{Office} = \text{`Paris'} \;\wedge\; \mathit{Budget} \geq 15000 \;\wedge\; \mathit{Budget} \leq 20000 \\
m_2 &\equiv \mathit{Office} \neq \text{`Paris'} \;\wedge\; \mathit{Budget} \geq 15000 \;\wedge\; \mathit{Budget} \leq 20000 \\
m_3 &\equiv \mathit{Office} = \text{`Paris'} \;\wedge\; \mathit{Budget} < 15000 \;\wedge\; \mathit{Budget} \leq 20000 \\
m_4 &\equiv \mathit{Office} \neq \text{`Paris'} \;\wedge\; \mathit{Budget} < 15000 \;\wedge\; \mathit{Budget} \leq 20000 \\
m_5 &\equiv \mathit{Office} = \text{`Paris'} \;\wedge\; \mathit{Budget} \geq 15000 \;\wedge\; \mathit{Budget} > 20000 \\
m_6 &\equiv \mathit{Office} \neq \text{`Paris'} \;\wedge\; \mathit{Budget} \geq 15000 \;\wedge\; \mathit{Budget} > 20000 \\
m_7 &\equiv \mathit{Office} = \text{`Paris'} \;\wedge\; \mathit{Budget} < 15000 \;\wedge\; \mathit{Budget} > 20000 \\
m_8 &\equiv \mathit{Office} \neq \text{`Paris'} \;\wedge\; \mathit{Budget} < 15000 \;\wedge\; \mathit{Budget} > 20000
\end{aligned}
$$

Horizontal Fragmentation: Prune Minterms

Step 3: Prune Set of Minterms

- Some constructed minterms may be unsatisfiable.
- Others can be simplified, because predicates imply one another.

Example:

$$
\begin{array}{lcl}
m_1 & \equiv & \mathit{Office} = \text{`Paris'} \;\wedge\; \mathit{Budget} \geq 15000 \;\wedge\; \mathit{Budget} \leq 20000 \\
m_2 & \equiv & \mathit{Office} \neq \text{`Paris'} \;\wedge\; \mathit{Budget} \geq 15000 \;\wedge\; \mathit{Budget} \leq 20000 \\
m_3 & \equiv & \mathit{Office} = \text{`Paris'} \;\wedge\; \mathit{Budget} < 15000 \;\cancel{\wedge\; \mathit{Budget} \leq 20000} \\
m_4 & \equiv & \mathit{Office} \neq \text{`Paris'} \;\wedge\; \mathit{Budget} < 15000 \;\cancel{\wedge\; \mathit{Budget} \leq 20000} \\
m_5 & \equiv & \mathit{Office} = \text{`Paris'} \;\cancel{\wedge\; \mathit{Budget} \geq 15000} \;\wedge\; \mathit{Budget} > 20000 \\
m_6 & \equiv & \mathit{Office} \neq \text{`Paris'} \;\cancel{\wedge\; \mathit{Budget} \geq 15000} \;\wedge\; \mathit{Budget} > 20000 \\
\cancel{m_7} & \cancel{\equiv} & \cancel{\mathit{Office} = \text{`Paris'} \;\wedge\; \mathit{Budget} < 15000 \;\wedge\; \mathit{Budget} > 20000} \\
\cancel{m_8} & \cancel{\equiv} & \cancel{\mathit{Office} \neq \text{`Paris'} \;\wedge\; \mathit{Budget} < 15000 \;\wedge\; \mathit{Budget} > 20000}
\end{array}
$$

Horizontal Fragmentation: Relevant Minterms

Step 4: Remove "Irrelevant" Predicates

- Enumeration leads to a large number of minterms ($\leadsto$ fragments).
  - $\rightarrow$ Each simple predicate breaks **all** fragments into two halves.
- Some simple predicates may not be a meaningful sub-fragmentation for all fragments.
  - $\rightarrow$ *E.g.*, a predicate might occur in the workload only in combination with another predicate.
- Thus: If two minterms $m_i = m \wedge p$ and $m_j = m \wedge \neg p$ are always accessed together (p is not relevant), drop p and replace m_i and m_j by just m.

(See Öszu and Valduriez; Principles of Distributed Database Systems; Springer 2011 for more details.)

Minterms → Fragments

Step 5: Define Fragments

Steps 1–4 resulted in a **set of minterms** (here: minterms m_1–m_6).

→ Each of these minterms defines one fragment.

$$R_1 \stackrel{\text{def}}{=} \sigma_{m_1}(R)$$
$$\vdots$$

→ Here: 6 fragments[24]

Note:

- We're still left with an **allocation strategy** to place fragments on (network) nodes.

[24] Some of these fragments may be empty for a given database instance. They are, nevertheless, fragments.

Derived Horizontal Fragmentation

Suppose we partitioned relation *Projects* horizontally.

- → To facilitate **joins**, it makes sense to **co-locate** tuples of *Projects* and *Employees*.
- → Define fragmentation of *Employees* based on fragmentation of *Projects*.

Projects			
pid	Title	Office	Budget
2	Eridanus	Paris	21000
3	Centaurus	Paris	17000
4	Andromeda	Rome	29000
1	Aquarius	London	16000
5	Pegasus	London	23000

Derived horizontal fragmentation:

$$\mathit{Employees}_{\mathit{Paris}} \stackrel{\text{def}}{=} \mathit{Employees} \ltimes \mathit{Projects}_{\mathit{Paris}}$$

- → To compute the join, it is now enough to consider only "corresponding" fragments.

Derived Horizontal Fragmentation

The correctness of primary horizontal fragmentations was easy to prove.

The correctness of **derived horizontal fragmentations** is less simple:

- **Completeness:**
  - → Employees that do not belong to any project will disappear.
  - → Completeness holds, however, when **referential integrity** is guaranteed.
- **Reconstruction:**
  - → The original relation can be re-constructed from a complete horizontal fragmentation using the **union** operator $\cup$.
- **Disjointness:**
  - → Semijoin operator $\ltimes$ does not prevent overlaps per se.
  - → Together with **integrity constraints**, disjointness may still be easy to show.

Vertical Fragmentation

Sometimes, it is more meaningful to split tables **vertically**:

Employees			
eid	Name	Proj.	Salary
628	J. Smith	1	58000
262	D. Miller	4	184000
381	P. Hanks	1	52000
725	D. Clark	3	55000
395	P. Jones	4	143000
738	S. Miles	2	38000

$\rightarrow$

Employees_1		
eid	Name	Proj.
628	J. Smith	1
262	D. Miller	4
381	P. Hanks	1
725	D. Clark	3
395	P. Jones	4
738	S. Miles	2

Employees_2	
eid	Salary
628	58000
262	184000
381	52000
725	55000
395	143000
738	38000

→ Keep key column in **both** fragments, so original relation can be re-assembled by means of a **join**.

→ Strictly speaking, vertical fragmentation always leads to **non-disjointness**.

Vertical Fragmentation

Finding a vertical fragmentation scheme is inherently more complex.

- "Only" 2^n minterms for n simple predicates.
- But $B(m)$ partitions for m non-key columns.[25]

Heuristics:

Group Create one fragment for each (non-key) column, then iteratively merge fragments.

Split Start with one relation and repeatedly partition it.

Input:

- Information about **attribute affinity**. Given two attributes A_i and A_j, how frequently are they accessed together in the workload?

[25] $B(m)$ is the mth Bell number; $B(10) \approx 115\,000$; $B(15) \approx 10^9$.

Hybrid Fragmentation

Horizontal and vertical fragmentation can be combined (arbitrarily).

E.g.,

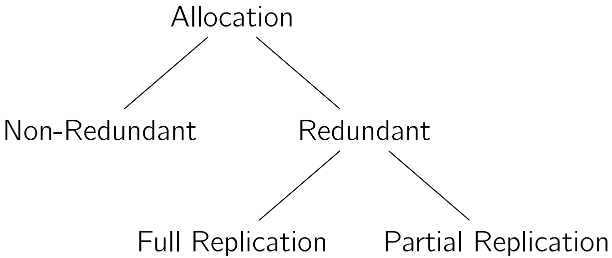
Employees$_{11}$		
eid	Name	Proj.
628	J. Smith	1
738	S. Miles	2
381	P. Hanks	1

Employees$_{12}$		
eid	Name	Proj.
725	D. Clark	3
395	P. Jones	4
262	D. Miller	4

Employees$_2$	
eid	Salary
628	58000
738	38000
381	52000
725	55000
395	143000
262	184000

→ Re-construct using a **combination of joins and unions**.

Next Step: Allocate fragments to nodes.



Replication

Replication is a two-edged sword:

	no replication	partial replication	full replication
query processing	hard	hard	easy
reliability	low	high	high
storage demand	low	moderate	high
parallel query potential	moderate	high	high
parallel update potential	high	moderate	low
concurrency control	easy	hard	moderate

Allocation — Criteria

Minimize Response Time

- Local data availability avoids communication delays.
- But updates might suffer from too much replication.

Maximize Availability

- Use redundancy to avoid down times.

Minimize Storage and Communication Cost

- For reads, replication may reduce communication; for writes it is the other way round.

Heuristic 1: "Non-Redundant Best Fit" Method

Rationale: What is the best node for each fragment?

1. **Analyze workload**: Which fragments are accessed by queries issued at which node?
   → Local placement **benefits** a query.
2. **Place each fragment** such that its **total benefit** is largest.
   → Break ties by allocating on the least loaded node.

Example: “Non-Redundant Best Fit”

fragment	accessed from node	number of accesses
R_1	H_1	12
	H_2	2
R_2	H_3	27
R_3	H_1	12
	H_2	12

→ Place fragment R_1 on node H_1.

→ Place fragment R_2 on node H_3.

→ Place fragment R_3 on node H_2 (H_1 already holds R_1).

"Non-Redundant Best Fit"

Pros:

- Easy to compute

Cons:

- Only considers benefits, but ignores costs
- Cannot support replication

Heuristic 2: "All Beneficial Nodes" Method

Rationale: Improve availability by allowing replication.

Placing a fragment R_i on a node H_j causes. . .

. . . a benefit:

- Improved **response time** for every query at H_j that references R_i.

. . . a cost:

- Effort to **update** the replica in case of writes.

Allocation strategy:

1. Compute, for all R_i/H_j combinations, the effective cost (cost minus benefit) of allocating R_i at H_j.
2. Place a fragment R_i on node H_j whenever benefit exceeds cost.

"All Beneficial Nodes" Method

Pros:

- Still simple

Cons:

- Network topology not considered (only local $\leftrightarrow$ remote)

Heuristic 3: “Progressive Fragment Allocation”

Rationale: Build on “All Beneficial Nodes”, but consider influence of allocation decisions on one another.

Strategy:

- Place one copy of each fragment so benefit/cost is maximised.
- Continue placing replicas one-by-one, always considering the existing fragment allocations.
  - → Stop when additional placement provides no more benefit.

Properties:

- Progressive Fragment Allocation considers the most relevant cost aspects at a reasonable algorithm complexity.

Query Processing over Fragmented Data

Consider an example:

```
SELECT p.Title
  FROM Employees AS e, Projects AS p
 WHERE e.Proj = p.pid
   AND e.Salary > 100000
```

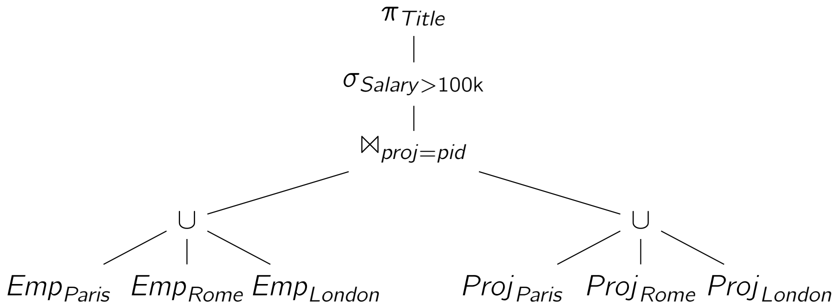
Let us assume

- *Projects* was **fragmented horizontally**, so project-relevant data can be stored local to the project;
- a **derived horizontal fragmentation** was used to co-locate employees with their projects.

What is a good way to execute the above join?

Re-Construct, Then Execute

Idea: Re-Construct global relations, then evaluate query:



$\rightarrow$ Use $\cup$ to re-construct horizontally fragmented relations.

Re-Construct, Then Execute

The resulting plan is **not very efficient**:

- Of both input relations all fragments except one must (at least) be sent over the network
  - $\rightarrow$ High **communication overhead**
  - $\rightarrow$ Index support?

However,

$$\left(R_1 \cup R_2\right) \bowtie \left(S_1 \cup S_2\right) = \left(R_1 \bowtie S_1\right) \cup \left(R_1 \bowtie S_2\right) \cup \left(R_2 \bowtie S_1\right) \cup \left(R_2 \bowtie S_2\right) \ .$$

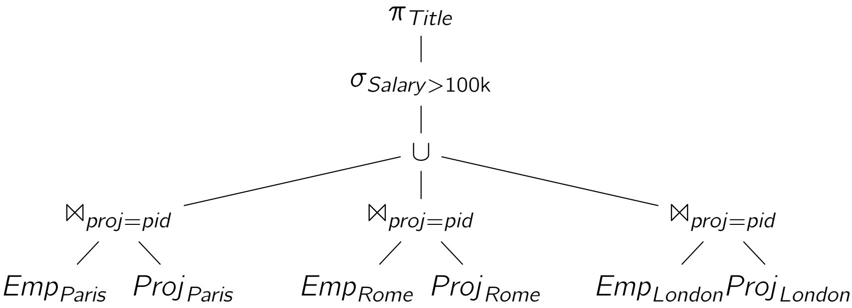
And, whenever $S_i = S \ltimes R_i$ (where $S = S_1 \cup \cdots \cup S_n$), then

$$R_i \bowtie S_j = \varnothing \qquad \text{for } i \neq j \ ,$$

such that

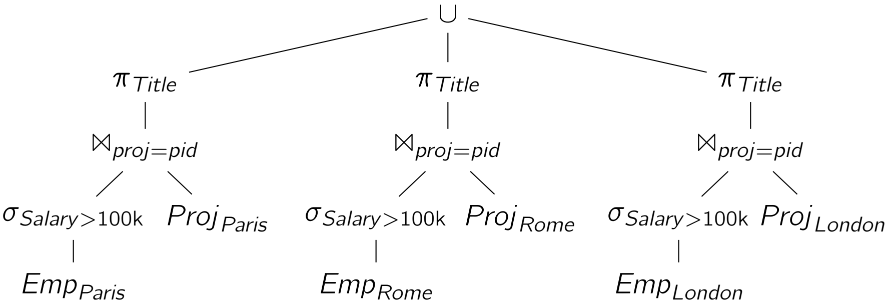
$$R \bowtie S = \left(R_1 \bowtie S_1\right) \cup \left(R_2 \bowtie S_2\right) \cup \cdots \cup \left(R_n \bowtie S_n\right) \ .$$

For the example, this leads to the (better) query plan



Re-Construct, Then Execute

Even better strategy: **push down projection and selection**:



$\rightarrow$ exploit (locally) **available indexes**

$\rightarrow$ reduce **transfer volume**

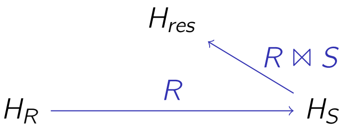
Join Queries in Distributed Databases

Generally, each join between two fragments could involve **three sites**:

- The fragment R is located on site H_R.
- The fragment S is located on site H_S.
- The result $R \bowtie S$ is needed on a third site H_{res}.

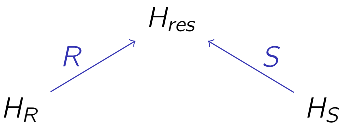
This leaves several simple **strategies to compute** $R \bowtie S$:

1. Send R to H_S, join on H_S, send result to H_{res}.

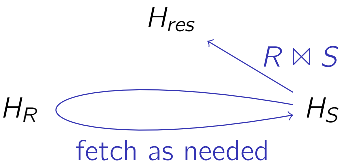


Simple Join Strategies

2. Finally, R and S could both be sent to H_{res} to compute the join there.



3. To avoid unnecessary transfers of R tuples to H_S, tuples could be fetched **on demand**.



Semi Join Filtering

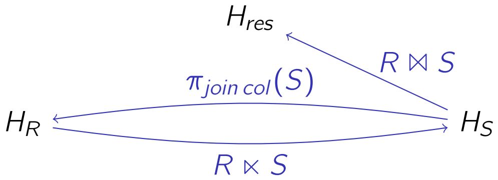
Rather than fetching R tuples one-by-one, why not fetch match candidates **in bulk**?

$\rightarrow$ Send list of join keys $H_S \rightarrow H_R$, reply with candidate list.

More formally, this can be achieved with help of **semi joins**:

$$R \bowtie S = \underbrace{\left(R \ltimes S\right)}_{\text{“candidate list”}} \bowtie S = \left(R \ltimes \underbrace{\pi_{join\,col}(S)}_{\text{“list of join keys”}}\right) \bowtie S$$

That is:



"Bloom Joins"

Once again, we can improve this idea by means of a **Bloom filter**.

→ Rather than sending $\pi_{join\,col} S$ along $H_S \rightarrow H_R$, send only a bit vector (Bloom filter).

→ Save **transfer volume** on the $H_S \rightarrow H_R$ link.

(False positives might slightly **increase** transfer volumes on the $H_R \rightarrow H_S$ link. But this increase is typically outweighed by savings along $H_S \rightarrow H_R$.)

Transaction Management

Distributed transactions may experience two new types of failure:

1. **partial system failure**
   - In a centralized system, all components fail or none at all.
   - In the distributed case, some nodes may fail, others may survive.
2. **network failure, network partitioning**
   - Nodes might seem dead, while in fact they're just in an unreachable network region.

To still guarantee ACID, we need protocols to ensure

- **atomic termination**;
- **global serialization**; and
- that **no global deadlocks** can happen.

Assumptions and Terminology

We assume the nodes in the system run independent database managers.

→ We refer to the database managers involved in a distributed transaction T as the **cohorts** of T.

We assume **each site supports ACID** and deadlock handling locally.

For each distributed transaction T there is one **coordinator**, *e.g.*,

→ dedicated coordinator

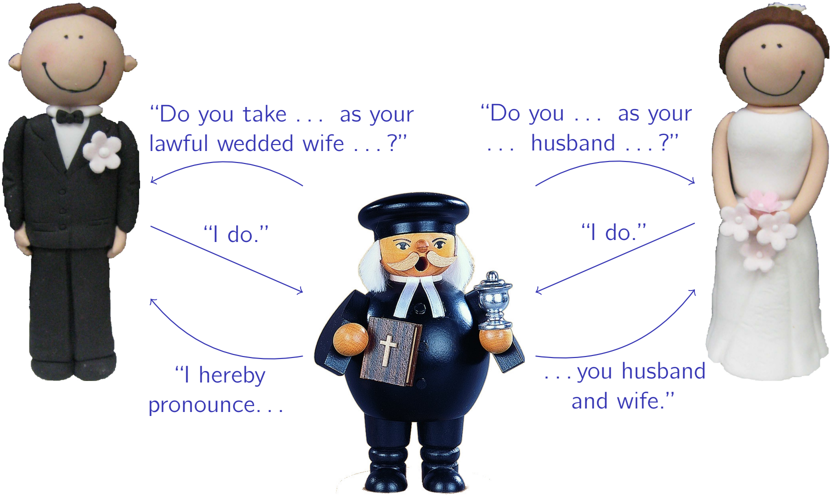
→ site where T was issued

→ elected coordinator, either once or per transaction.

Atomic Commit Protocol

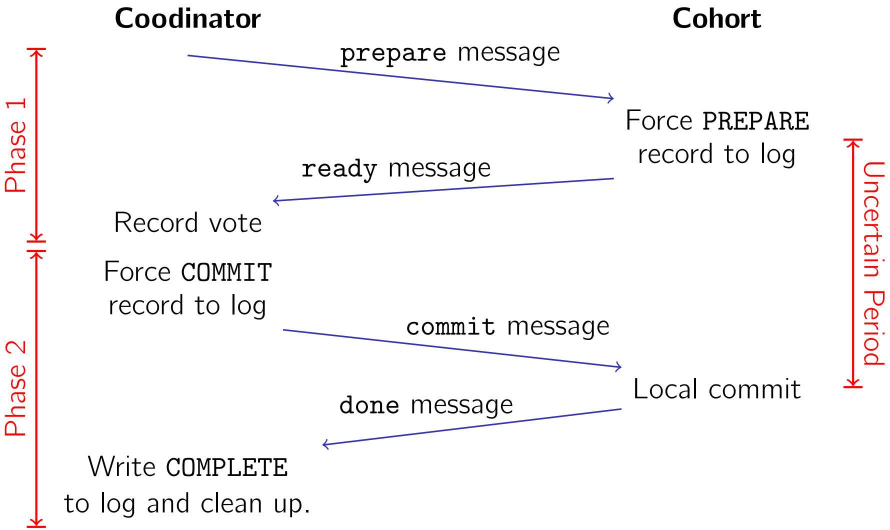
Cohorts must reach **agreement** on the outcome of a transaction.

→ Every cohort must have the **chance to veto/abort**.



Two-Phase Commit Protocol

The **two-phase commit protocol** follows the same principle:



Two-Phase Commit Protocol

1. Coordinator sends `prepare` message to all cohorts.
2. If a cohort is **willing to commit**:
   - → Respond with `ready`.
   - → Confirms that cohort is **able to commit** (even if it crashes after response) → force `PREPARE` to log.
   - → Cohort **cannot unilaterally abort** after sending `ready`.
   - → After sending `ready`, cohort waits for `commit` from coordinator.

   Otherwise, the cohort responds with `abort`.

After sending `ready`, the cohort must **wait** for the coordinator decision.

- → Cannot commit locally, yet. Other cohorts might have voted `abort`.
- → Cannot abort locally—promised to coordinator that it won't.

Two-Phase Commit Protocol

3. Coordinator receives and records each cohort's vote.
4. Coordinator decides whether TX can be **committed globally**.
   → **commit:** Force `COMMIT` to log, then send `commit` to all cohorts.
   → **abort:** Send `ABORT` to all cohorts.
5. Upon `COMMIT`, cohorts commit locally and respond with `done`.
6. After all cohorts have responded `done`, coordinator can release its data structures for this transaction.

✎ **Which is the point that actually marks the TX as committed?**

Dealing with Failures—Timeouts

Timeout Protocol:

- Triggered when a site does not receive an expected message.

Cohort times out while **waiting for `prepare` message**.

- No global decision made, yet.
- Cohort can unilaterally decide on **abort**.
  - → Respond to later `prepare` with `abort`.

Coordinator times out while **waiting for `ready`/`abort` vote**.

- Similar situation, can decide on **abort**.

Dealing with Failures—Timeouts

Cohort times out while **waiting for** `commit/abort` **message**.

- **Cannot** unilaterally decide on **commit** or **abort**.
- Only option: Try to determine transaction outcome.
  - → Actively request from coordinator (which might be unreachable).
  - → Ask other cohorts.
    (If another cohort hasn't voted yet, both can decide to abort.)
- Otherwise the cohort remains **blocked**.

Coordinator times out while **waiting for** `done` **message**.

- Not a critical situation. Coordinator just cannot release its resources.

Dealing with Failures—Machine Crashes

Restart Protocol:

- Triggered when coordinator or cohort restart after a crash.

Coordinator Restart:

- `COMMIT` record found in log:
  - → Send `commit` to all cohorts
    (Crash might have happened before `commits` were sent.)
- No `COMMIT` record found in log:
  - → Protocol was still in phase 1 when crash occured.
  - → Coordinator had decided on abort before crash.
  - → In both cases: **abort** transaction (by sending `abort`).

Dealing with Failures—Machine Crashes

Cohort Restart:

- `COMMIT` record found in log:
  - → Local commit completed successfully. Nothing more to do.
- `PREPARE` record found in log:
  - → Must request TX outcome (from coordinator).
- No `PREPARE` record found in log:
  - → No commitment made to coordinator.
  - → Can decide on **abort** unilaterally.

Global Serialization

To ensure **serializability**:

- Manage locks at **central site** $\rightsquigarrow$ **centralized concurrency control**
  - $\rightarrow$ Single point of failure
  - $\rightarrow$ High communication overhead

    Local transactions must go through (remote) lock manager, too!

- Manage locks **local to the data**
  - $\rightarrow$ Global serializability?

Global Serialization

Theorem:

Locally: **strict two-phase locking**

$\Downarrow$ **Two-Phase Commit**

Global schedule is serializable.

$\rightarrow$ Local serializability plus two-phase commit are enough to realize global serializability.

Distributed Deadlocks

Some strategies for **deadlock handling** also work in distributed settings:

- **asymmetric lock handling:** wait-die/wound-wait
- **timeout**

Distributed **deadlock detection** is more difficult:

- Periodically collect waits-for information at a **central site**.
  - $\rightarrow$ Then handle as in single-machine case.
  - $\rightarrow$ Might cause high network transfer volumes.
- When a deadlock is suspected, try to **discover** it through peer-to-peer information exchange.
  - $\rightarrow$ T waits for a lock on an external site H $\rightarrow$ contact H.

Data Replication—Read-One/Write-All

Replication:

→ Improve **availability** (possibly also efficiency)

How guarantee consistency?

Strategy 1: Synchronous replication; read-one/write-all

- **Writes** are synchronously propagated to **all** replica sites.
  - → **Lock** at least one replica immediately; lock and update all at commit time.
  - → Coordinate replica updates, *e.g.*, using Two-Phase Commit.
- **Reads** may use **any** replica.

→ Good for **read-heavy** workloads.

→ Lots of locks → locking overhead, risk of deadlocks

→ Writes cannot complete when a replica site is unavailable.

Data Replication—Quorum Consensus Protocol

Strategy 2: Synchronous replication; Quorum Consensus Protocol

Problem:

- A reader does not see a write's change, because both looked at different replica of the same object.

Thus:

- Make sure readers and writers always "see" one another.
  $\rightarrow$ in "read-one/write-all" this was guaranteed.

Quorum Consensus Protocol

Quorum Consensus Protocol:

- Total number of replica (of some item): N
- **Readers** access at least Q_R copies.
- **Writers** access at least Q_W copies.

To detect read/write conflicts:

$\rightarrow$ Read set/write set must **overlap**.

$\rightarrow$ $Q_R + Q_W > N$

To detect write/write conflicts:

$\rightarrow$ Write set/write set must **overlap**.

$\rightarrow$ $Q_W + Q_W > N \quad (\Leftrightarrow 2 \cdot Q_W > N)$

Quorum Consensus Protocol

Protocol can be **tuned** to trade update cost $\leftrightarrow$ availability.

- Read-one/write-all: $Q_R = 1$; $Q_W = N$

Implementation:

- Store **commit time stamp** with each object.
  $\rightarrow$ Use the latest version within the read object set.
- Node unavailability is not a problem, as long as transactions can assemble necessary quorums.

Variant of Quorum Consensus:

- Set a **weight** w_i for each replica.
- Quorums must now satisfy $Q_R + Q_W > \sum_i w_i$ and $2 \cdot Q_W > \sum_i w_i$.

Asynchronous Replication: Primay Copy Replication

Strategy 3: Asynchronous replication; primary copy

- For each object, one replica is designated its **primary copy**.
- All **updates** go to primary copy.
- Updates are **propagated asynchronously** to secondary copies.
- **Reads** go to any node.

Properties:

→ Asynchronous replication avoids high overhead at commit time.

→ Simple to implement: Forward **write-ahead log** to secondary copies.

→ Good fit for many application patterns

Primary Copy Replication

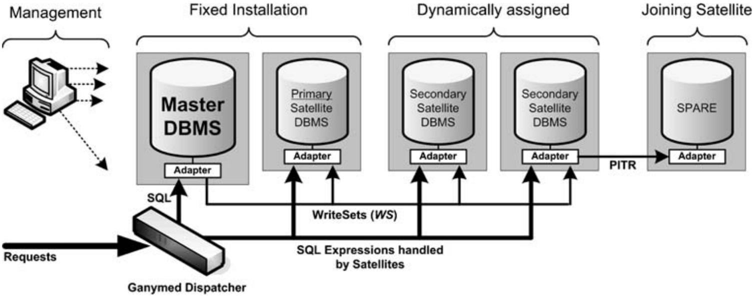
However:

- Reader might see **old/inconsistent data**.

Guarantee Serializability:

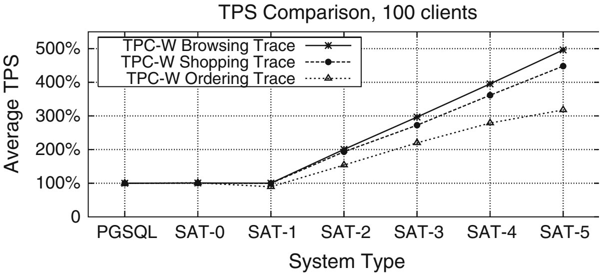
- Run **read-only** transactions on secondary copy sites.
- Run **read/write** transactions on primary copy site.
  - $\rightarrow$ Reads of read/write transactions go to primary site, too.
  - $\rightarrow$ Alternative: Readers **wait** on secondary sites if necessary.
- **Multi-version concurrency control** $\rightarrow$ consistent reads.

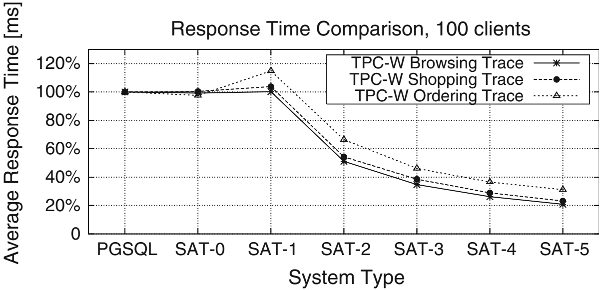
Example: Ganymed



Plattner *et al.* Extending DBMSs with Satellite Databases. *VLDB Journal*, 17:657–682, 2008.

Example: Ganymed





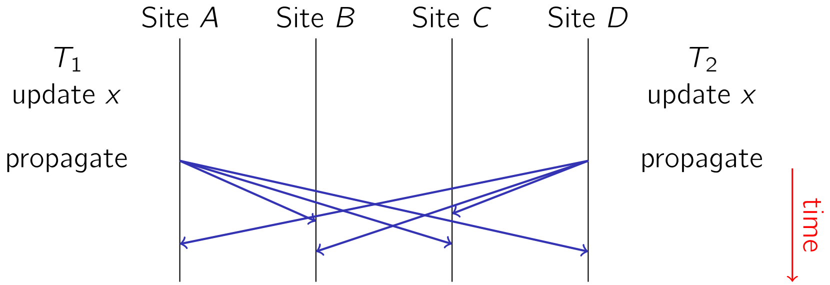
Scenarios

Scenarios for asynchronous replication:

- **Data Warehousing:**
  → Propagate changes from transactional system to warehouse (*e.g.*, periodically).
- **Specialized Satellites:**
  → Satellite systems need **not** be identical to primary copy.
  → Build specialized **indexes** on satellites.
  → Use different **data organization** (*e.g.* column store)
  → etc.
- **Hot Standby:**
  → Secondary provides an up-to-date copy of all data.
  → Swap primary ↔ secondary in case of failure.

Strategy 3: Asynchronous replication; group replication

- Allow updates on **any** replica and propagate afterward.



→ **Consistency?**

Group Replication

Conflicting updates might arrive at a site.

- Need a **conflict resolution** mechanism.
- *E.g.*, assign **time stamps** to updates and let latest win.
  - → Replicas will **eventually** contain the same value.
  - → **No serializability**, however.
    (*E.g.*, **lost updates** are still possible.)
- Sometimes, **user-defined conflict resolution** makes sense.
  - → *E.g.*, accumulate value increments.

Brewer's CAP Theorem

We've seen multiple **trade-offs** between

- **Consistency**
  In the database domain, we'd like to have ACID guarantees.
- **Availability**
  Every request received by a non-failing node should result in a response.
- **Partition Tolerance**
  No set of failures less than a total network outage should cause the system to respond incorrectly.

Brewer's CAP Theorem

In a PODC keynote 2000, Eric Brewer stated the **"CAP Theorem"**:

> In a distributed computer system it is **impossible** to provide **all of the three guarantees**
>
> - **Consistency**,
> - **Availability**, and
> - **Partition Tolerance**.

Notes:

- Here, "consistency" means "linearizability," a criterion usually used in the distributed systems community.

CAP Theorem

Two of the three CAP properties can be achieved together:

- **Consistency and Availability** (drop Partition Tolerance)
  Many of the techniques we discussed with provide consistency and availability, but they will fail when a partition happens.
- **Consistency and Partition Tolerance** (drop Availability)
  E.g., always enforce consistency; deny service when nodes do not respond.
- **Availability and Partition Tolerance** (drop Consistency)
  System might become inconsistent when a partition happens; *e.g.*, "group replication" discussed above.

CAP Theorem—Proof

Proof by contradiction; assume

- System provides **all three properties**.
- Two nodes G_1 and G_2 in **separate partitions**
  - $\rightarrow$ G_1 and G_2 cannot communicate.

Initially, the value of v is v_0 on all nodes.

1. A **write** occurs on G_1, updating $v_0 \rightarrow v_1$.
   - $\rightarrow$ By the availability assumption, this write completes.
2. A **later read** occurs on G_2.
   - $\rightarrow$ Read will complete (availability), but return **old value** v_0.

$\lightning$ Consistency is violated.
(Or, to ensure consistency, either the read or the write would have to block because of the network partition.)

Consequences

So, since we cannot have all three...

...drop partition tolerance?

→ What does this mean?
We can try to improve network reliability; but partitions might **still** occur. And if a partition happens, what will be the consequence?

...drop availability?

→ A (generally) unavailable system is useless.

→ In practice: loss of availability $\equiv$ loss of money.

...drop consistency?

→ DB people really don't like to give up consistency. ☺

→ Yet, it's best understood and can typically be handled.

Consequences

Trade-off:

availability $\leftrightarrow$ consistency ?

- Systems that sacrifice consistency tend to do so **all the time**.
- Availability only given up **when partitioning happens**.

Many systems, strictly speaking, even give up **both!**

$\rightarrow$ Improve **latency** by doing so.

BASE Properties

Many large-scale distributed systems follow the **BASE** principles:

- **B**asically **A**vailable,
  - $\rightarrow$ Prioritize availability
- **S**oft State,
  - $\rightarrow$ Data might change (without user input); *e.g.*, to reach consistency.
- **E**ventually Consistent.
  - $\rightarrow$ System might be inconsistent at times, but "eventually" reach a consistent state ($\rightsquigarrow$ group replication)

An example of the (new) availability $\leftrightarrow$ consistency trade-off is **Amazon's Dynamo**[26].

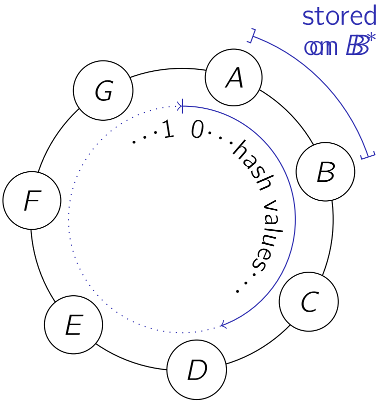
Situation at Amazon:

- Service-oriented architecture, decentralized
  - $\rightarrow$ Page request results in $\approx$ 150 service requests.
  - $\rightarrow$ Need **stringent latency bounds** ($\leadsto$ look at 99.9th percentile).
- **Availability** is top priority
  - $\rightarrow$ Everything else is a lost selling opportunity.
  - $\rightarrow$ CAP theorem: "drop consistency"
  - $\rightarrow$ Choose **asynchronous replication**, **no primary copy**
  - $\rightarrow$ Need **conflict resolution** strategy

[26] DeCandida *et al.* Dynamo: Amazon's Highly Availabe Key-value Store. *SOSP '07*.

Fragmentation and Allocation in Dynamo

- Hash all key values into the range $[0, 1[$ ($\rightsquigarrow$ treat as a **ring**).
- Nodes are placed at random positions $[0, 1[$.
- Place an object $o = \langle k, v \rangle$ on the node that follows $hash(k)$ clockwise.
  - → Place on next N nodes for replication factor N.
- When a node H joins/leaves:
  - → Copy data from/to node that precedes/follows H.



*stored on B, C, D if replication factor is 3

Consistent Hashing; Virtual Nodes

Advantages:

- Resilience to skew
- Easy to scale (add/remove nodes to ring)

Problem:

- Hot spot when a node joins/leaves, or in case of node failure.

Thus:

- Let each **physical machine** represent **multiple nodes** in the ring ($\rightsquigarrow$ "virtual nodes"); position all (virtual) nodes randomly in the ring.
  - $\rightarrow$ Every (physical) machines neighbors with multiple others.
  - $\rightarrow$ Avoid hot spots.
  - $\rightarrow$ Stronger hardware $\rightarrow$ more positions in the ring.

Replication and Consistency

Dynamo uses a variant of **quorum consensus** to realize replication.

- Starting from an object o's hash value $hash(k)$, the first N (virtual) nodes that follow clockwise hold replicas of o.[27]
- These N nodes are called the **preference list** for k.
- Read/write objects according to quorums Q_R/Q_W ($\nearrow$ slide 445).
- Use Q_R and Q_W to tune for application needs.
  - $\rightarrow$ Typical values: $N = 3$, $Q_R = Q_W = 2$.
  - $\rightarrow$ Read-mostly applications: $Q_W = N$, $Q_R = 1$.

[27] Actually, chooose replica nodes such that replicas end up on different machines.

Hinted Handoff

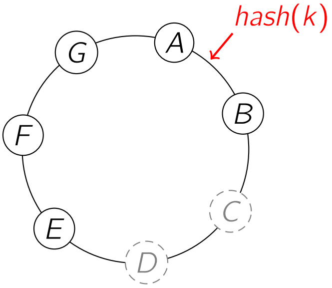
Problem: Quorum may be unreachable because of **failures**.

$\rightarrow$ "partition tolerance"

Thus: Use first N **healthy** nodes for read/write operations.

E.g.,

- Quorum: $N = 3$; $Q_R = Q_W = 2$
- Key h hashes between A and B
- C and D are unavailable
- Send write to B, E, and F.
  - $\rightarrow$ The latter two with a **hint**
  - $\rightarrow$ E and F will attempt to deliver the update to C and D.



Inconsistencies / Conflict Resolution

Hinted handoff may lead to **inconsistencies**.

Conflict resolution: Latest update wins?

$\rightarrow$ Risk of **lost updates** ($\nearrow$ slide 454)

Thus:

- Track **causality** and resolve conflicts automatically.
  - $\leadsto$ syntactic reconciliation
- Otherwise defer conflict resolution to **application**.
  - $\leadsto$ semantic reconciliation

Data Versioning / Vector Clocks

Data Versioning:

- With each stored object, keep **version information**.
- Version information: **vector of timestamp counters** $\mathbf{x} = (x_1, \dots, x_k)$
  - → One vector position for each node in the system
  - → "vector clock"
  - → In practice, implement vector as list of $\langle node, counter \rangle$ pairs.
- Multiple versions of the same object may be in the system at the same time.
  - → A `get()` operation returns all of them, together with their vector clock.
  - → Reconcile them after the read; generate new vector clock; and write back new version.

Read/Write Operations

E.g., read/write combination executed on node m:

/* Read (all) old versions */

$\{\langle \mathbf{x}_1, \mathit{value_1}\rangle, \langle \mathbf{x}_2, \mathit{value_2}\rangle, \dots, \langle \mathbf{x}_n, \mathit{value_n}\rangle\} \leftarrow$ `get` (*key*) ;

/* Reconcile */

$\langle \mathbf{x}, \mathit{value}\rangle \leftarrow$ `reconcile` ($\{\langle \mathbf{x}_1, \mathit{value_1}\rangle, \dots, \langle \mathbf{x}_n, \mathit{value_n}\rangle\}$) ;

/* Increment vector clock **x** at position m */

$\mathbf{x}[m] \leftarrow \mathbf{x}[m] + 1$;

/* Write back new version (with new vector clock **x**) */

`put` (**x** , *key* , *value*) ;

Causality:

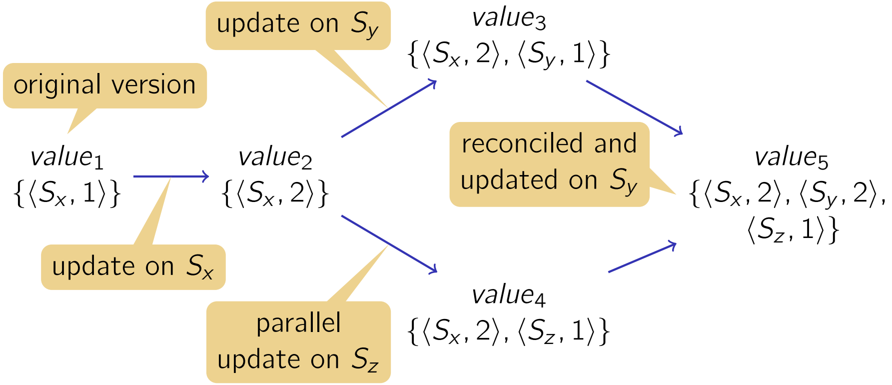
- Given two vector clocks $\mathbf{x} = (x_1, \ldots, x_k)$ and $\mathbf{y} = (y_1, \ldots, y_k)$,

$$\forall i = 1, \ldots, k : x_i \leq y_i \quad \Rightarrow \quad \mathbf{x} \twoheadrightarrow \mathbf{y} \ ,$$

  i.e., **y descends** from **x**.

- $\mathbf{x} \twoheadrightarrow \mathbf{y}$ means there is a **causal relation** from **x** to **y**.
  → **x** "older" than **y** and can be discarded (syntactic reconciliation).
- If neither $\mathbf{x} \twoheadrightarrow \mathbf{y}$ nor $\mathbf{y} \twoheadrightarrow \mathbf{x}$, they are a result of **parallel updates**.
  → Semantic reconciliation necessary.
- **New vector clock**: Use $\max(x_i, y_i)$ for each vector position i.

Vector Clocks: Example



Conflict detected during last update:

- Node S_y reads *value*$_3$ and *value*$_4$ with their version clocks.

Implementation Issues

Coordinators:

- Choose a "coordinator" to handle update of an object o.
  - $\rightarrow$ One of the nodes in o's **preference list**.
- Dynamo lives in a **trusted environment**.
  - $\rightarrow$ Link storage node interaction direct into client application.

Vector Clocks:

- Few coordinators for every object o
  - $\rightarrow$ Version vector sparse (most counters are 0)
  - $\rightarrow$ Implement as **list of** $\langle \mathit{node}, \mathit{counter} \rangle$ **pairs**
- Vector sizes will grow over time
  - $\rightarrow$ **Limit** number of list entries (*e.g.*, 10 entries)
  - $\rightarrow$ **Truncate vector clocks** if necessary

Vector Clock Truncation

In practice, parallel/conflicting versions are rare

→ Truncating vector clocks won't actually hurt.

E.g., Live trace over 24 hours at Amazon:

- 99.94 % requests saw 1 version
- 0.00057 % saw 2 versions
- 0.00047 % saw 3 versions
- 0.00009 % saw 4 versions

Ring Membership / Replica Synchronization

Ring Membership:

- Propagate membership information through **gossip-based protocol**.
  - → Avoid single point of failure
  - → Node arrival or departure announced explicitly in Dynamo

Replicas might still go **out of sync**:

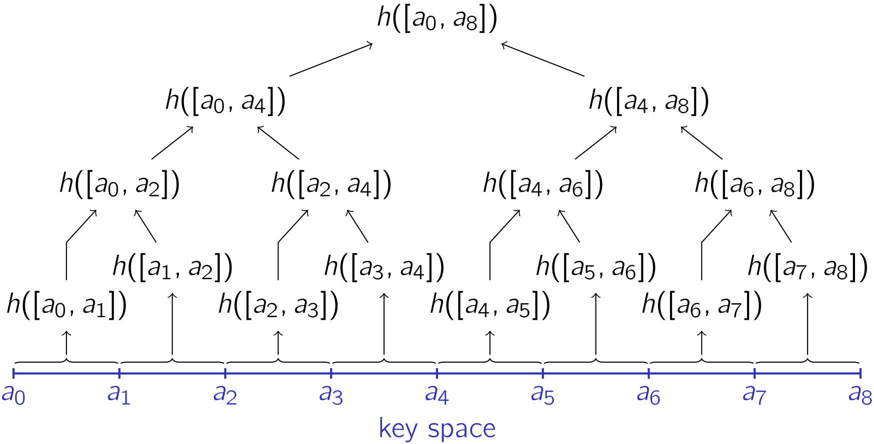
- *E.g.*, hinted handoff: backup node goes offline before it can forward updates to final destination (↗ slide 466)

Use **Merkle trees** to check/re-establish consistency:

- Only little data exchange necessary to locate inconsistencies.

Merkle Trees

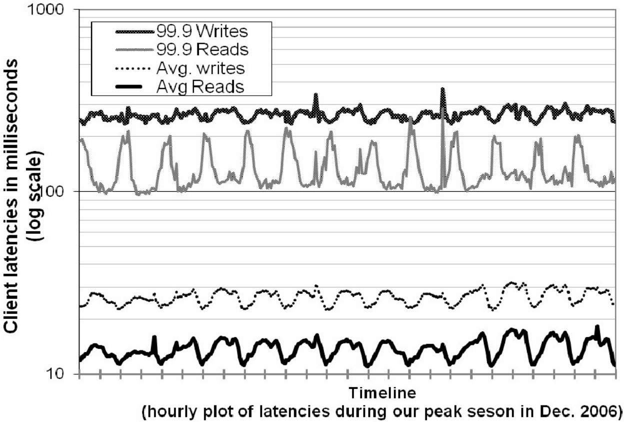
Tree of hashes, which cover the key space below them:



Dynamo Performance

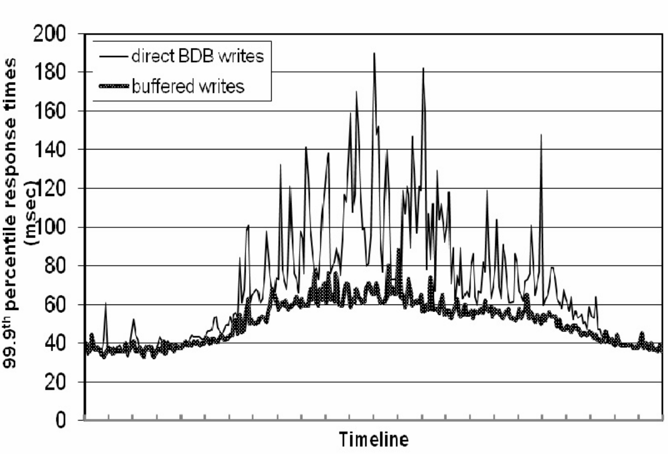
Performance criterion:

- Strong **latency guarantees**
- *e.g.*, SLA: 99.9 % of all requests must execute within 300 ms.
  - → Average performance is **not** the primary criterion here.



Dynamo Performance

Buffered Writes: trade durability ↔ performance



Compromise: Force flush on only one node (out of the Q_W).

Partitioning and Placement on Storage Nodes

Strategy 1: (as discussed before)

- Place (virtual) nodes randomly in key space
  - → Partitioning and placement are intertwined.
- Simple to scale on paper, harder to do in practice:
  - → Data must be moved when nodes are added/removed
  - → Since partitioning changes, everything has to be re-computed: data to move, Merkle trees, etc.
  - → Data archival for changing key ranges?

Partitioning and Placement on Storage Nodes

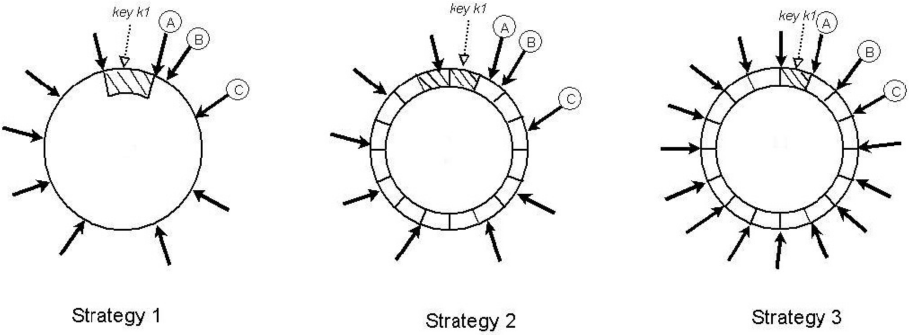
Strategy 2: (equi-sized partitions; random tokens for each storage node)

- Generate random ring positions for each (virtual) node, as before.
- **Static partitioning**; Q equi-sized partitions.
  → Use partition end to determine preference list
  → All keys in one partition reside on same node

Strategy 3: (deployed meanwhile at Amazon)

- Equi-sized partitions; assign partitions (randomly) to nodes.
  → Randomly distribute/ "steal" partitions when a node leaves/joins.
- Partitioning now simple and fixed
  → Data structures for one partition don't change, just have to be moved (*e.g.*, when nodes leave/join, or for backup).
  → Membership information more compact to represent.

Partitioning/Placement Strategies



Evaluation of Partitioning/Placement Strategies