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
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.

<table>
<thead>
<tr>
<th>Projects</th>
<th>Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>Title</td>
</tr>
<tr>
<td>1</td>
<td>Aquarius</td>
</tr>
<tr>
<td>2</td>
<td>Eridanus</td>
</tr>
<tr>
<td>3</td>
<td>Centaurus</td>
</tr>
<tr>
<td>4</td>
<td>Andromeda</td>
</tr>
<tr>
<td>5</td>
<td>Pegasus</td>
</tr>
</tbody>
</table>

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

- \( Projects_1 = \sigma_{Office='Paris'}(Projects) \)
- \( Projects_2 = \sigma_{Office='Rome'}(Projects) \)
- \( Projects_3 = \sigma_{Office='London'}(Projects) \)
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 $R_i$.
  $\rightarrow$ “It must be possible to define a relational operator $\nabla$ such that $R = \nabla (R_1, \ldots, R_n)$.”

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

$$R \leadsto R_1 = \sigma_{\bar{p}}(R) ; \hspace{1cm} R_2 = \sigma_{\neg p}(R) .$$
Idea: Derive \( p_i \) from workload information.

Step 1: Analyze workload

- **Qualitative Information:** Predicates used in queries
  - Extract **simple predicates** of the form
    \[ s_j = \text{attribute } \theta \text{ constant} \]
    where \( \theta \in \{=, <, \neq, \leq, >, \geq\} \).
  - Observe that simple predicates are easy to negate.
  - We refer to a conjunction of (negated) simple predicates as a **minterm**.

- **Quantitative Information:**
  - **minterm selectivity**
  - **access frequency** (of a minterm or a query)
Example

Queries:

\[ Q_1: \]
\[
\text{SELECT Title}
\]
\[
\text{FROM Projects}
\]
\[
\text{WHERE Office = 'Paris'}
\]

\[ Q_2: \]
\[
\text{SELECT Office}
\]
\[
\text{FROM Projects}
\]
\[
\text{WHERE Budget BETWEEN 15000 AND 20000}
\]

Simple Predicates:

- \( s_1 \equiv \text{Office = 'Paris'} \)
- \( s_2 \equiv \text{Budget} \geq 15000 \)
- \( s_3 \equiv \text{Budget} \leq 20000 \)
Step 2: Enumerate Possible Minterms

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

Example:

\[
\begin{align*}
    m_1 &\equiv \text{Office} = \text{‘Paris’} \land \text{Budget} \geq 15000 \land \text{Budget} \leq 20000 \\
    m_2 &\equiv \text{Office} \neq \text{‘Paris’} \land \text{Budget} \geq 15000 \land \text{Budget} \leq 20000 \\
    m_3 &\equiv \text{Office} = \text{‘Paris’} \land \text{Budget} < 15000 \land \text{Budget} \leq 20000 \\
    m_4 &\equiv \text{Office} \neq \text{‘Paris’} \land \text{Budget} < 15000 \land \text{Budget} \leq 20000 \\
    m_5 &\equiv \text{Office} = \text{‘Paris’} \land \text{Budget} \geq 15000 \land \text{Budget} > 20000 \\
    m_6 &\equiv \text{Office} \neq \text{‘Paris’} \land \text{Budget} \geq 15000 \land \text{Budget} > 20000 \\
    m_7 &\equiv \text{Office} = \text{‘Paris’} \land \text{Budget} < 15000 \land \text{Budget} > 20000 \\
    m_8 &\equiv \text{Office} \neq \text{‘Paris’} \land \text{Budget} < 15000 \land \text{Budget} > 20000
\end{align*}
\]
Step 3: Prune Set of Minterms

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

Example:

\[ m_1 \equiv Office = 'Paris' \land Budget \geq 15000 \land Budget \leq 20000 \]
\[ m_2 \equiv Office \neq 'Paris' \land Budget \geq 15000 \land Budget \leq 20000 \]
\[ m_3 \equiv Office = 'Paris' \land Budget < 15000 \land Budget \leq 20000 \]
\[ m_4 \equiv Office \neq 'Paris' \land Budget < 15000 \land Budget \leq 20000 \]
\[ m_5 \equiv Office = 'Paris' \land Budget \geq 15000 \land Budget > 20000 \]
\[ m_6 \equiv Office \neq 'Paris' \land Budget \geq 15000 \land Budget > 20000 \]
\[ m_7 \equiv Office = 'Paris' \land Budget < 15000 \land Budget > 20000 \]
\[ m_8 \equiv Office \neq 'Paris' \land Budget < 15000 \land Budget > 20000 \]
Step 4: Remove “Irrelevant” Predicates

- Enumeration leads to a large number of minterms (fragments).
  - Each simple predicate breaks all fragments into two halves.
- Some simple predicates may not be a meaningful sub-fragmentation for all fragments.
  - E.g., a predicate might occur in the workload only in combination with another predicate.
- Thus: If two minterms \( m_i = m \land p \) and \( m_j = m \land \neg p \) 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.)
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.

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

→ 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.
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*.

**Derived** horizontal fragmentation:

\[
\text{Employees}_{Paris} \overset{\text{def}}{=} \text{Employees} \times \text{Projects}_{Paris}
\]

→ To compute the join, it is now enough to consider only “corresponding” fragments.
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 $\bowtie$ 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**:

<table>
<thead>
<tr>
<th>Employees</th>
<th>Employees$_1$</th>
<th>Employees$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>eid</strong></td>
<td><strong>Name</strong></td>
<td><strong>Proj.</strong></td>
</tr>
<tr>
<td>628</td>
<td>J. Smith</td>
<td>1</td>
</tr>
<tr>
<td>262</td>
<td>D. Miller</td>
<td>4</td>
</tr>
<tr>
<td>381</td>
<td>P. Hanks</td>
<td>1</td>
</tr>
<tr>
<td>725</td>
<td>D. Clark</td>
<td>3</td>
</tr>
<tr>
<td>395</td>
<td>P. Jones</td>
<td>4</td>
</tr>
<tr>
<td>738</td>
<td>S. Miles</td>
<td>2</td>
</tr>
</tbody>
</table>

→ 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 $m$th Bell number; $B(10) \approx 115,000$; $B(15) \approx 10^9$. 
Hybrid Fragmentation

Horizontal and vertical fragmentation can be combined (arbitrarily).

E.g.,

\[
\begin{array}{|c|c|c|}
\hline
\text{Employees}_{11} & & \\
\hline
\text{eid} & \text{Name} & \text{Proj.} \\
\hline
628 & J. Smith & 1 \\
738 & S. Miles & 2 \\
381 & P. Hanks & 1 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{Employees}_{12} & \text{Salary} \\
\hline
\text{eid} & \\
\hline
725 & 58000 \\
395 & 38000 \\
262 & 52000 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{Employees}_{11} & & \\
\hline
\text{eid} & \text{Name} & \text{Proj.} \\
\hline
628 & J. Smith & 1 \\
738 & S. Miles & 2 \\
381 & P. Hanks & 1 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{Employees}_{12} & \text{Salary} \\
\hline
\text{eid} & \\
\hline
725 & 58000 \\
395 & 38000 \\
262 & 52000 \\
\hline
\end{array}
\]

→ Re-construct using a **combination of joins and unions**.
Next Step: Allocate fragments to nodes.

- Allocation
  - Non-Redundant
    - Full Replication
  - Redundant
    - Partial Replication
Replication is a two-edged sword:

<table>
<thead>
<tr>
<th></th>
<th>no replication</th>
<th>partial replication</th>
<th>full replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>query processing</td>
<td>hard</td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td>reliability</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>storage demand</td>
<td>low</td>
<td>moderate</td>
<td>high</td>
</tr>
<tr>
<td>parallel query potential</td>
<td>moderate</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>parallel update potential</td>
<td>high</td>
<td>moderate</td>
<td>low</td>
</tr>
<tr>
<td>concurrency control</td>
<td>easy</td>
<td>hard</td>
<td>moderate</td>
</tr>
</tbody>
</table>
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.
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”

<table>
<thead>
<tr>
<th>fragment</th>
<th>accessed from node</th>
<th>number of accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>$H_1$</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>$H_2$</td>
<td>2</td>
</tr>
<tr>
<td>$R_2$</td>
<td>$H_3$</td>
<td>27</td>
</tr>
<tr>
<td>$R_3$</td>
<td>$H_1$</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>$H_2$</td>
<td>12</td>
</tr>
</tbody>
</table>

→ 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 ↔ 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.
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?**
Idea: Re-Construct global relations, then evaluate query:

\[ \pi_{\text{Title}} \]

\[ \sigma_{\text{Salary} > 100k} \]

\[ \Join_{\text{proj} = \text{pid}} \]

\[ \bigcup \text{Emp}_{\text{Paris}} \bigcup \text{Emp}_{\text{Rome}} \bigcup \text{Emp}_{\text{London}} \]

\[ \bigcup \text{Proj}_{\text{Paris}} \bigcup \text{Proj}_{\text{Rome}} \bigcup \text{Proj}_{\text{London}} \]

→ Use \( \bigcup \) 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
  - High communication overhead
  - Index support?

However,

\[
(R_1 \cup R_2) \Join (S_1 \cup S_2) = (R_1 \Join S_1) \cup (R_1 \Join S_2) \cup (R_2 \Join S_1) \cup (R_2 \Join S_2).
\]

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

\[
R_i \Join S_j = \emptyset \quad \text{for } i \neq j,
\]

such that

\[
R \Join S = (R_1 \Join S_1) \cup (R_2 \Join S_2) \cup \cdots \cup (R_n \Join S_n).
\]
Re-Construct, Then Execute

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

\[
\pi \text{Title} \\
\sigma \text{Salary} > 100k \\
\bigcup \\
\forall \text{proj} = \text{pid} \\
\forall \text{proj} = \text{pid} \\
\forall \text{proj} = \text{pid}
\]

\[
\text{Emp}_{Paris} \\
\text{Proj}_{Paris} \\
\text{Emp}_{Rome} \\
\text{Proj}_{Rome} \\
\text{Emp}_{London} \\
\text{Proj}_{London}
\]
Even better strategy: **push down projection and selection:**

\[
\begin{align*}
\pi_{\text{Title}} \\
\land_{\text{proj}=\text{pid}} \\
\sigma_{\text{Salary}>100k} \\
\text{Emp}_{\text{Paris}} & \quad \text{Proj}_{\text{Paris}} \\
\end{align*}
\]

\[
\begin{align*}
\pi_{\text{Title}} \\
\land_{\text{proj}=\text{pid}} \\
\sigma_{\text{Salary}>100k} \\
\text{Emp}_{\text{Rome}} & \quad \text{Proj}_{\text{Rome}} \\
\end{align*}
\]

\[
\begin{align*}
\pi_{\text{Title}} \\
\land_{\text{proj}=\text{pid}} \\
\sigma_{\text{Salary}>100k} \\
\text{Emp}_{\text{London}} & \quad \text{Proj}_{\text{London}} \\
\end{align*}
\]

\[\cup\]

→ exploit (locally) **available indexes**

→ 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 $H_{res}$.

![Diagram showing the join process](image)
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?

→ 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:

$$R \bowtie S = (R \times S) \bowtie S = (R \times \pi_{\text{join col}}(S)) \bowtie S$$

“candidate list”  “list of join keys”

That is:

$$H_{res}$$

$H_R$  $\pi_{\text{join col}}(S)$  $R \bowtie S$

$H_S$  $R \times S$
“Bloom Joins”

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

→ Rather than sending $\pi_{\text{join\_col}}S$ 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 $H_S \rightarrow H_R$.)
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**.

“Do you take . . . as your lawful wedded wife . . .?”

“I do.”

“I hereby pronounce. . . .

“Do you . . . as your . . . husband . . .?”

“I do.”

. . . you husband and wife.”
The **two-phase commit protocol** follows the same principle:

- **Coordinator**
  - prepare message
  - ready message
  - Force COMMIT record to log
  - commit message
  - done message
  - Write COMPLETE to log and clean up.

- **Cohort**
  - Force PREPARE record to log
  - Local commit

---

© Jens Teubner · Architecture & Implementation of DBMS · Summer 2018
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 occurred.
  → 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** → **centralized concurrency control**
  - Single point of failure
  - High communication overhead
  - Local transactions must go through (remote) lock manager, too!

- Manage locks **local to the data**
  - Global serializability?
Theorem:

Locally:  **strict two-phase locking**

arrow

Two-Phase Commit

Global schedule is serializable.

→ 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**.
  - Then handle as in single-machine case.
  - Might cause high network transfer volumes.
- When a deadlock is suspected, try to **discover** it through peer-to-peer information exchange.
  - $T$ waits for a lock on an external site $H$ → 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.
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.
  → in “read-one/write-all” this was guaranteed.
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:**
- Read set/write set must **overlap**.
- $Q_R + Q_W > N$

**To detect write/write conflicts:**
- Write set/write set must **overlap**.
- $Q_W + Q_W > N$  ($\iff 2 \cdot Q_W > N$)
Quorum Consensus Protocol

Protocol can be **tuned** to trade update cost ↔ availability.

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

Implementation:

- Store **commit time stamp** with each object.
  - 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 $2 \cdot Q_W > \sum_i w_i$. 
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.
  - Reads of read/write transactions go to primary site, too.
  - Alternative: Readers **wait** on secondary sites if necessary.
- **Multi-version concurrency control** → consistent reads.
The problem of these designs is that they assume that the middleware layer can control everything happening inside the database engine. This is not accurate assumption (concurrency control affects more than just tables, e.g., recovery procedures, indexes). For these approaches to work correctly, functionality such as triggers, user-defined functions and views would have to be disabled or the concurrency control at the middleware level would have to work at an extremely conservative level. In the same spirit, Ganymed imposes no data organization, structuring of the load, or particular arrangements of the schema (unlike, e.g., [18]).

In terms of the DBMSs that the architecture should support, the objective is flexibility and, thus, we do not rely on engine specific functionality. The design we propose does not rely on the existence of special features or modifications to the underlying DBMS.

2.2 Overview

The system works by routing transactions through a dispatcher over a set of backend databases. For a given dispatcher, the backends consist of one master and a set of satellites. The objective is to use the satellites to extend the master according to two principles: clients see a consistent database at all times and the master can take over if the satellites cannot deal with a query. The latter point is crucial to understanding the design of the system. In the worst case, our system behaves as a single database: the master. When in doubt, the dispatcher routes the traffic to the master. We also rely on the master to provide industrial strength (e.g., crash recovery and fault tolerance). The idea is that the satellites extend the functionality or capacity of the master but neither replace it nor implement redundant functionality. This same principle applies to the problem of replicating triggers, user-defined functions, etc. Our system is not meant to extend that functionality. Thus, transactions that involve triggers or user-defined functions are simply sent to the master for execution there.

As a assumption we make is that we can achieve a perfect partition of the load between master and satellites. However, unlike previous work [10, 26], we do not require the data to be manually partitioned across nodes. For the purposes of this paper, the loads we consider involve full replication and specialized functionality (skyline queries and keyword search). For fully replicated satellites, the master executes all write operations while the satellites execute only queries (read-only transactions). In the case of specialized functionality the satellites execute skyline queries and keyword searches, all other transactions are done at the master. We also assume that queries can be answered within a single satellite.

2.3 Main components

The main components of the system are as follows (see Fig. 2). The dispatcher is responsible for routing transactions to the master and satellites. It acts as front end for clients. The system is controlled and administered from a management console. Communication with the backend DBMSs always takes place through adapters, which are thin layers of software installed on the DBMS machines.

In terms of database machines, we consider three types: masters, primary satellites, and secondary satellites. Primary satellites are optional and used for dynamic creation of satellites. The purpose of primary satellites is to be able to create new satellites without hitting the master for all the data necessary to spawn a new satellite. When implementing dynamic satellites, there is always one primary satellite attached to the master. Secondary satellites are those created dynamically.

PITR stands for the technique we use in our prototype to dynamically create satellites. Please refer to Sect. 8.

Extending DBMSs with satellite databases

6.5.1 Performance and scalability

The first part of the evaluation analyzes performance and scalability. The Ganymed prototype was compared with a reference system consisting of a single PostgreSQL instance. We measured the performance of the Ganymed dispatcher in different configurations, from 0 up to 5 satellites. This gives a total of seven experimental setups (called PGSQL and SAT-0 ≤ n ≤ 5), each setup was tested with the three different TPC-W traces.

The load generator was then attached to the database (either the single instance database or the dispatcher, depending on the experiment). During a measurement interval of 100 s, a trace was then fed into the system over 100 parallel client connections and at the same time average throughput and response times were measured. All transactions, read only and updates, were executed in the SERIALIZABLE mode. Every experiment was repeated until a sufficient, small standard deviation was reached.

Figure 6 shows the results for the achieved throughput (transactions per second) and average transaction response times, respectively. The ratio of aborted transactions was below 0.5% for all experiments.

Figure 7 shows two example histograms for the TPC-W ordering mix workload: on the left side the reference system, on the right side SAT-5. The sharp drop in performance in the SAT-5 histogram is due to multiple PostgreSQL replicas that did checkpointing of the WAL (write ahead log) at the same time. The replicas were configured to perform this process at least every 300 s; this is the default for PostgreSQL.

Based on the graphs, we can prove the lightweight structure of the Ganymed prototype. In a relay configuration, where only one replica is attached to the Ganymed dispatcher, the achieved performance is almost identical to the PostgreSQL reference system. The performance of the setup with two replicas, where one replica is used for updates and the other for read-only transactions, is comparable to the single replica setup. This clearly reflects the fact that the heavy part of the TPC-W loads consists of complex read-only queries. In the case of the write intensive TPC-W ordering mix, a two replica setup is slightly slower than the single replica setup. In the setups where more than two replicas are used, the performance compared to the reference system could be significantly improved. A close look at the response times chart shows that they converge. This is due to the RSI-PC algorithm which uses parallelism for different transactions, but no intra-parallelism for single transactions.

© Jens Teubner · Architecture & Implementation of DBMS · Summer 2018
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 4: Asynchronous replication; group replication

- Allow updates on any replica and propagate afterward.

\[ T_1 \]
update \( x \)

propagate

\[ T_2 \]
update \( x \)

propagate

→ Consistency?
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.
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.
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** $v_0$.

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

- **Basically Available,**
  - Prioritize availability
- **Soft State,**
  - Data might change (without user input); *e.g.*, to reach consistency.
- **Eventually Consistent.**
  - System might be inconsistent at times, but “eventually” reach a consistent state (group replication)
Amazon Dynamo

An example of the (new) availability ↔ consistency trade-off is Amazon’s Dynamo\textsuperscript{26}.

Situation at Amazon:

- Service-oriented architecture, decentralized
  - Page request results in $\approx 150$ service requests.
  - Need **stringent latency bounds** ($\sim$ look at 99.9\textsuperscript{th} percentile).

- **Availability** is top priority
  - Everything else is a lost selling opportunity.
  - CAP theorem: “drop consistency”
  - Choose **asynchronous replication, no primary copy**
  - Need **conflict resolution** strategy

\textsuperscript{26}DeCandida et al. Dynamo: Amazon’s Highly Available Key-value Store. *SOSP ‘07.*
Hash all key values into the range \([0, 1]\) (\(\sim\) 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.

\(\rightarrow\) Place on next \(N\) nodes for replication factor \(N\).

When a node \(H\) joins/leaves:

\(\rightarrow\) Copy data from/to node that precedes/follows \(H\).
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 (\(\sim\) “virtual nodes”); position all (virtual) nodes randomly in the ring.
  - Every (physical) machines neighbors with multiple others.
  - Avoid hot spots.
  - Stronger hardware \(\rightarrow\) more positions in the ring.
Dynamo uses a variant of *quorum consensus* to realize replication.

- Starting from an object o’s hash value $\text{hash}(k)$, 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 $Q_R/Q_W$ (↗ slide 445).
- Use $Q_R$ and $Q_W$ to tune for application needs.
  - Typical values: $N = 3$, $Q_R = Q_W = 2$.
  - Read-mostly applications: $Q_W = N$, $Q_R = 1$.

$^{27}$Actually, choose replica nodes such that replicas end up on different machines.
**Problem:** Quorum may be unreachable because of **failures**.

→ “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*.
  → The latter two with a **hint**
  → *E* and *F* will attempt to deliver the update to *C* and *D*.  

© Jens Teubner · Architecture & Implementation of DBMS · Summer 2018
Hinted handoff may lead to **inconsistencies**.

**Conflict resolution:** Latest update wins?

→ Risk of **lost updates** (↗ slide 454)

Thus:

- Track **causality** and resolve conflicts automatically.
  - syntactic reconciliation
- Otherwise defer conflict resolution to **application**.
  - semantic reconciliation
Data Versioning / Vector Clocks

Data Versioning:

- With each stored object, keep **version information**.
- Version information: **vector of timestamp counters** \( x = (x_1, \ldots, x_k) \)
  - One vector position for each node in the system
  - “vector clock”
- 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 $m$:

```plaintext
/* Read (all) old versions */  
1 $\{\langle x_1, value_1 \rangle, \langle x_2, value_2 \rangle, \ldots, \langle x_n, value_n \rangle\} \leftarrow \text{get} (key) ;$
/* Reconcile */
2 $\langle x, value \rangle \leftarrow \text{reconcile} (\{\langle x_1, value_1 \rangle, \ldots, \langle x_n, value_n \rangle\}) ;$
/* Increment vector clock $x$ at position $m$ */
3 $x[m] \leftarrow x[m] + 1 ;$
/* Write back new version (with new vector clock $x$) */
4 $\text{put} (x, key, value) ;$
```
Reconciliation

Causality:

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

$$\forall i = 1, \ldots, k : x_i \leq y_i \implies \mathbf{x} \rightarrow \mathbf{y},$$

i.e., $\mathbf{y}$ descends from $\mathbf{x}$.

- $\mathbf{x} \rightarrow \mathbf{y}$ means there is a causal relation from $\mathbf{x}$ to $\mathbf{y}$.
  - $\rightarrow \mathbf{x}$ “older” than $\mathbf{y}$ and can be discarded (syntactic reconciliation).

- If neither $\mathbf{x} \rightarrow \mathbf{y}$ nor $\mathbf{y} \rightarrow \mathbf{x}$, they are a result of parallel updates.
  - Semantic reconciliation necessary.

- New vector clock: Use $\max(x_i, y_i)$ for each vector position $i$. 
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 $o$.
  - One of the nodes in $o$’s **preference list**.
- Dynamo lives in a **trusted environment**.
  - Link storage node interaction directly into client application.

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

- 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:

```
h([a_0, a_8])
```

```
h([a_0, a_4])
```

```
h([a_2, a_4])
```

```
h([a_0, a_2])
```

```
h([a_1, a_2])
```

```
h([a_0, a_1])
```

```
h([a_3, a_4])
```

```
h([a_2, a_3])
```

```
h([a_2, a_3])
```

```
h([a_4, a_5])
```

```
h([a_4, a_5])
```

```
h([a_4, a_6])
```

```
h([a_5, a_6])
```

```
h([a_6, a_7])
```

```
h([a_6, a_7])
```

```
h([a_6, a_8])
```

```
h([a_7, a_8])
```

```
h([a_0, a_8])
```

```
h([a_0, a_4])
```

```
h([a_2, a_4])
```

```
h([a_4, a_8])
```

```
h([a_4, a_8])
```

```
h([a_6, a_8])
```

```
h([a_6, a_8])
```

```
h([a_7, a_8])
```

```
h([a_0, a_8])
```

```
h([a_0, a_4])
```

```
h([a_2, a_4])
```

```
h([a_4, a_8])
```

```
h([a_4, a_8])
```

```
h([a_6, a_8])
```

```
h([a_6, a_8])
```

```
h([a_7, a_8])
```

```
h([a_0, a_8])
```

```
h([a_0, a_4])
```

```
h([a_2, a_4])
```

```
h([a_4, a_8])
```

```
h([a_4, a_8])
```

```
h([a_6, a_8])
```

```
h([a_6, a_8])
```

```
h([a_7, a_8])
```

```
h([a_0, a_8])
```

```
h([a_0, a_4])
```

```
h([a_2, a_4])
```

```
h([a_4, a_8])
```

```
h([a_4, a_8])
```

```
h([a_6, a_8])
```

```
h([a_6, a_8])
```

```
h([a_7, a_8])
```
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 \( Q_W \)).
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

**Strategy 1**

**Strategy 2**

**Strategy 3**
Evaluation of Partitioning/Placement Strategies

The typical configuration deployed for majority of our services. The different strategies for system with 30 nodes and N=3 with Figure 8: Comparison of the recovery. (ii)

Since partition ranges are fixed, they can be stored in separate addition to this, strategy 3 is advantageous and simpler to deploy gossip the membership information periodically and as such it is orders of magnitude. While storage size of membership information ma...

Dynamo instances from using Strate...

worst load balancing efficiency. For a brief time, Strategy 2 achieves the best load balancing e...

The results are given in Figure 8. As seen in the figure, strategy 3 evaluate the skew in their load distribution while all strategies use the same amount of space to maintain their membership information. For instance, in strategy 1 each node needs to maintain the information regarding...

strategy 3 each node needs to maintain the information regarding...

the partitions assigned to each node.

In our next experiment, these strategies were evaluated by varying the partitions assigned to each node.

For example, in strategy 1 each node needs to maintain the token positions of all the nodes in the ring and in strategy 3 each node needs to maintain the information regarding...

Archiving the entire dataset stored by Dynamo is simpler in...