Demystifying the Zoo of Contemporary Database Systems

CS165 Section
Niv Dayan
Introduction
Introduction

• Different architectures
  – Performance
  – Data integrity
  – User interface
Introduction

• Different architectures
  – Performance
  – Data integrity
  – User interface
Introduction

• **Theme:** any trend in database technology can be traced to a trend in hardware

  ![Database designer](image1.png) ![Hardware](image2.png)
  
  Database designer    Hardware

• **Claim:** The new database technologies are adaptations to changes in hardware
DBHistory
History

• 3 goals of database design
  – Speed
  – Affordability
  – Resilience to system failure

• How you achieve them depends on hardware
History

• Two storage media:
  
  Main Memory
  Fast, expensive, volatile

  Disk
  Slow, cheap, non-volatile
History

• How should data be stored across them?
• Main memory is volatile and expensive
History

- To make a system fast, address bottleneck
- Disk is extremely slow
History

• To make a system fast, address bottleneck
• Disk is extremely slow
History

• Why so slow?

• Two questions:
  – Question 1: How to minimize disk access?
  – Question 2: What to do during a disk access?
**History**

- **Problem**: How to minimize disk accesses?
- **Solution**: Store data that is frequently co-accessed at the same physical location
- Consolidates many disk accesses to one
**History**

- **Example:** Bank
- Co-locate all information about each customer
- Customer Sara deposits $100

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Will</td>
<td>450</td>
</tr>
</tbody>
</table>

2 disk accesses, since data about Sara is co-located

---

Main Memory

Disk

Database

Add 100

Example: Bank
History

• What to do during a disk access?
• Start running the next operation(s)
• Improves performance
• But data can get corrupted
History

• A couple, Bob and Sara, share a bank account
• Both deposit $100 at same time

balance = 0
History

• A couple, Bob and Sara, share a bank account
• Both deposit $100 at same time

\[ \text{balance} = 0 \]

Retrieve

\[ \text{balance} = 0 \]

Fetch
History

• A couple, Bob and Sara, share a bank account
• Both deposit $100 at same time

\[ \text{balance} = 0 \]
History

• A couple, Bob and Sara, share a bank account
• Both deposit $100 at same time
History

• A couple, Bob and Sara, share a bank account
• Both deposit $100 at same time
A couple, Bob and Sara, share a bank account. Both deposit $100 at the same time.
History

- A couple, Bob and Sara, share a bank account
- Both deposit $100 at same time
History

- A couple, Bob and Sara, share a bank account
- Both deposit $100 at same time

balance = 100
History

• A couple, Bob and Sara, share a bank account
• Both deposit $100 at same time

Account balance should be 200!
Bob and Sara lost money.

balance = 100
History

• **Question:** how to achieve concurrency while maintaining data integrity?

• **Insight:** transactions can be concurrent, as long as they don’t modify the same data

• **Solution:** locking
  – Bob locks data, modifies it, releases lock
  – Sara waits until lock is released

• **Downside:**
  – transactions may need to wait for locks.
History

• 3 goals of database design
  – Speed
  – Affordability
  – Resilience to system failure
History

• 3 goals of database design
  – Speed
  – Affordability
  – Resilience to system failure
History

- Disk was cheap, but not so cheap
- 1 gigabyte for $10000 in 1980
- Avoid storing replicas of same data

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>
History

- **Solution:** “Normalization”. Break tables.

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>

**Customers**

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
</tr>
</tbody>
</table>

**Accounts**

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>

- **Bonus:** Easier to maintain data integrity
History

- **Normalization:**
  - Saves storage space
  - Easier to maintain data integrity
- **Downside:** reads are more expensive
  - Need to join tables

<table>
<thead>
<tr>
<th>Customers</th>
<th>Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
</tr>
</tbody>
</table>
History

- Data is decomposed across tables
- Query Language: SQL
  - select balance from Customers c, Accounts a
    where c.account-ID = a.ID and c.name = "Bob"

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>
History

• 3 goals of database design
  – Speed
  – Affordability
  – Resilience to system failure
History

• 3 goals of database design
  – Speed
  – Affordability
  – Resilience to system failure
History

• Many things can go wrong
  – Power failure
  – Hardware failure
  – Natural disaster

• Data is precious (e.g. bank)

• Provide recovery mechanism
History

- **Example**: Sara transfers $100 to Anna
- Power stops in the middle

Sara’s balance = 0

Sara’s balance = 100

Anna’s balance = 450
History

• **Example:** Sara transfers $100 to Anna
• Power stops in the middle

Sara’s balance = 0

Sara’s balance = 100

Anna’s balance = 450
History

- **Example:** Sara transfers $100 to Anna
- Power stops in the middle

Sara’s balance = 0

Anna’s balance = 550
History

- **Example**: Sara transfers $100 to Anna
- Power stops in the middle

At this point, power fails

Anna’s balance = 550

Sara’s balance = 0

Anna’s balance = 450
History

- **Transaction**: a sequence of operations all takes place, or none take place.
- Transactions should be atomic
History

• **Problem:** how to guarantee atomicity?
• **Solution:** use a log (on disk)
• All data changes are recorded in the log
• After power failure, examine log
• Undo changes by unfinished transactions
History

• Data integrity
  – Concurrency (fix with locking)
  – System failure (fix with logging)

• ACID
  – Atomicity
  – Consistency
  – Isolation
  – Durability
History

• Summary
  – Speed
  – Affordability
  – Resilience to system failure

• Relational databases:
  – Normalize data into multiple tables
  – ACID (locking & logging)
  – SQL

• Design decisions are motivated by hardware
Today
Today

• What changed in hardware?
• How does it affect database design?
Today

- Disk is $10^7$ times cheaper
- Main memory is $10^6$ times cheaper
Today

• Disk is now dirt cheap
• Organizations keep all historical data
• Business intelligence
• E.g. Amazon
  – revenues from product X on date Y
  – which products are bought together
Today

- **Traditional system architecture:**

  - End users
  - Transactional Database
  - Business Intelligence
Today

• **Problem:**
  – Analytical queries are expensive
    • Touch a lot of data
    • Disk access
    • Locks
  – They slow down transactions.
  – End-users wait longer
Today

• **Solution**: split database
Today

• Different workloads
• Different internal design
• Example analytical queries
  – How long is delivery? (2 columns, all rows)
  – Revenue from product X? (2 columns, all rows)

• Problem:
  – Data is stored row by row
• **Solution: column-store**
  - Each column is stored separately
  - Good for analytical queries
  - Changes entire architecture
  - Examples: Vertica, Vectorwise, Greenplum, etc.
Today

• How are transactional databases affected by hardware changes?
Today

- Main memory is cheaper
- Terabytes are affordable
- Enough to store all transactional data
- E.g. Amazon
  - Products list
  - User accounts
Today

- Main memory was expensive
- Now it’s cheaper
Today

- Transactional databases are main memory databases
- Bottleneck used to be disk access
- The new bottleneck is ACID (logging, locking)
Today

- More challenges
- Due to internet, 100% availability is key
- Data is replicated
Today

• Joins become more expensive

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>
Today

- Replication and locks become more expensive

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>
Today

• **Single machine bottlenecks:**

  ![Database](image1.png)
  - Logging & locking

• **Multiple machine bottlenecks:**

  ![Database](image2.png)
  - Replication, locks, joins
Today

• NoSQL and NewSQL address these
  – NoSQL simplifies
  – NewSQL engineers
NoSQL
(MongoDB)
NoSQL
(MongoDB)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Dec 2016</th>
<th>Nov 2016</th>
<th>Dec 2015</th>
<th>DBMS</th>
<th>Database Model</th>
<th>Score</th>
<th>Score</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dec 2016</td>
<td>Nov 2016</td>
<td>Dec 2015</td>
</tr>
<tr>
<td>1.</td>
<td>1.</td>
<td>1.</td>
<td>1.</td>
<td>Oracle</td>
<td>Relational DBMS</td>
<td>1404.40</td>
<td>-8.60</td>
<td>-93.15</td>
</tr>
<tr>
<td>2.</td>
<td>2.</td>
<td>2.</td>
<td>2.</td>
<td>MySQL</td>
<td>Relational DBMS</td>
<td>1374.41</td>
<td>+0.85</td>
<td>+75.87</td>
</tr>
<tr>
<td>3.</td>
<td>3.</td>
<td>3.</td>
<td>3.</td>
<td>Microsoft SQL Server</td>
<td>Relational DBMS</td>
<td>1226.66</td>
<td>+12.86</td>
<td>+103.50</td>
</tr>
<tr>
<td>4.</td>
<td>4.</td>
<td>↑5.</td>
<td></td>
<td>PostgreSQL</td>
<td>Relational DBMS</td>
<td>330.02</td>
<td>+4.20</td>
<td>+49.92</td>
</tr>
<tr>
<td>5.</td>
<td>5.</td>
<td>↓4.</td>
<td></td>
<td>MongoDB</td>
<td>Document store</td>
<td>328.68</td>
<td>+3.21</td>
<td>+27.29</td>
</tr>
<tr>
<td>6.</td>
<td>6.</td>
<td></td>
<td>6.</td>
<td>DB2</td>
<td>Relational DBMS</td>
<td>184.34</td>
<td>+2.89</td>
<td>-11.78</td>
</tr>
<tr>
<td>7.</td>
<td>7.</td>
<td>↑8.</td>
<td></td>
<td>Cassandra</td>
<td>Wide column store</td>
<td>134.28</td>
<td>+0.31</td>
<td>+3.44</td>
</tr>
<tr>
<td>8.</td>
<td>8.</td>
<td>↓7.</td>
<td></td>
<td>Microsoft Access</td>
<td>Relational DBMS</td>
<td>124.70</td>
<td>-1.27</td>
<td>-15.51</td>
</tr>
<tr>
<td>9.</td>
<td>9.</td>
<td>↑10.</td>
<td></td>
<td>Redis</td>
<td>Key-value store</td>
<td>119.89</td>
<td>+4.35</td>
<td>+19.36</td>
</tr>
<tr>
<td>10.</td>
<td>10.</td>
<td>↓9.</td>
<td></td>
<td>SQLite</td>
<td>Relational DBMS</td>
<td>110.83</td>
<td>-1.17</td>
<td>+9.98</td>
</tr>
</tbody>
</table>

http://db-engines.com/en/ranking
NoSQL

• Name popularized in 2009
• Conference on “open source distributed non-relational databases”
• NoSQL was a hashtag
NoSQL

• Different types

  – Document stores
  – Column-oriented
  – Key-value-stores

  – Graph databases

Similar
  – MongoDB
  – Cassandra
  – Redis

Different
  – Neo4j
NoSQL

• MongoDB - Main decisions

1. No joins
   • Aggregate related data into “documents”
   • Reduces network traffic
   • Data modeling is harder

2. No ACID
   • Faster
   • Concurrency & system failure can corrupt data
NoSQL

• Single machine bottlenecks:
  - Logging & locking

• Multiple machine bottlenecks:
  - Replication, locks, joins
NoSQL

• To avoid joins, data is de-normalized

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>
NoSQL

- In MongoDB

Collection of customer Documents

```javascript
{ name: "Bob", account-ID: 1, balance: 100 }
{ name: "Sara", account-ID: 1, balance: 100 }
{ name: "Trudy", account-ID: 2, balance: 450 }
```

- `db.customers.find(name: "Sara")`
NoSQL

• Documents are flexible

```json
{    name: "Bob",
    account-ID: 1,
    balance: 100,
    favorite-color: "red",
    credit-score: 3.0
}
```

```json
{    name: "Sara",
    account-ID: 1,
    balance: 100,
    hobbies: ["rowing", "running"]
}
```
NoSQL

• Main point: no need for joins
• All related data is in one place

```
{  name: "Bob",
   account-ID: 1,
   balance: 100,
   favorite-color: "red",
   credit-score: 3.0
}
```
NoSQL

- Single machine bottlenecks:

- Multiple machine bottlenecks:

  - Logging & locking
  - Replication, locks, joins
NoSQL

- MongoDB does not lock
- Recall Bob and Sara
- Deposit $100 at same time to shared account
- Overwrite each other’s update

No general way to prevent this
NoSQL

• Single machine bottlenecks:

  Logging & locking

• Multiple machine bottlenecks:

  Replication, locks, joins
NoSQL

- Eventual consistency
- Different operation order across replicas
- E.g. concurrent addition and multiplication

Deposit
Add 100

Interest
Multiply by 1.1

{ ...
  balance: 100
  ...
}

{ ...
  balance: 100
  ...
}
NoSQL

• Eventual consistency
• Different operation order across replicas
• E.g. concurrent addition and multiplication

Deposit
Add 100

Interest
Multiply by 1.1

{ ... balance: 100 ...
  ... }

{ ... balance: 100 ...
  ... }
NoSQL

- Eventual consistency
- Different operation order across replicas
- E.g. concurrent addition and multiplication

Deposit
Add 100

Interest
Multiply by 1.1

Inconsistent replicas

{ ... balance: 220 ... }

{ ... balance: 210 ... }
NoSQL

• Single machine bottlenecks:
  
  Logging & locking

• Multiple machine bottlenecks:
  
  Replication, locks, joins
NoSQL

• When to use MongoDB?
• Non-interacting entities
  – No sharing (e.g. bank account)
  – No exchanging (e.g. money transfers)
• Commutative operations on data
• You need a flexible data model
NewSQL
NewSQL

- ACID & good performance
- Redesign internal architecture.
NewSQL

• Data is normalized into multiple tables
• Tables partitioned and replicated across machines

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Trudy</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>
NewSQL

- Single machine bottlenecks:
  - Logging & locking

- Multiple machines bottlenecks:
  - Replication, locks, joins
NewSQL

• **History**: concurrent transactions were introduced since disk was slow
• **Today**: Now all data is in main memory
  • Transactions in main memory are fast
  • Less need for concurrency

• VoltDB removes concurrency
• Thus, no need for locking
NewSQL

- Recall Bob and Sara
- Deposit $100 at same time to shared account
- Overwrite each other’s update

In VoltDB, this cannot happen
NewSQL

- **Single machine bottlenecks:**
  - Replication, locks, joins
  - **Logging & locking**

- **Multiple machine bottlenecks:**
  - Replication, locks, joins
NewSQL

• **History**: log introduced for recovery
• **Today**: it takes too long to recover from log
• Instead, replicate data across machines
• If one machine fails, others continue working
• Simplifies logging
NewSQL

- Single machine bottlenecks:
  - Logging & locking

- Multiple machine bottlenecks:
  - Replication, locks, joins
NewSQL

- Try to avoid joins across machines
- Store data that is commonly accessed at same time on same machine

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Trudy</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>450</td>
</tr>
</tbody>
</table>
NewSQL

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>account-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sara</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

- Alleviates problem
- Does not solve it (e.g. money transfer)
NewSQL

• Single machine bottlenecks:

  - Logging & locking

• Multiple machine bottlenecks:

  - Replication, locks, joins
NewSQL

- Tables are replicated
- Enforce operation order across replicas

Deposit
Add 100

Interest
Multiply by 1.1

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>100</td>
</tr>
</tbody>
</table>
NewSQL

- Tables are replicated
- Enforce operation order across replicas

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

Deposit
Add 100

Interest
Multiply by 1.1
NewSQL

- Tables are replicated
- Enforce operation order across replicas

Deposit
Add 100

Interest
Multiply by 1.1

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>200</td>
</tr>
</tbody>
</table>
NewSQL

• Tables are replicated
• Enforce operation order across replicas

Deposit
Add 100

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>200</td>
</tr>
</tbody>
</table>

Interest
Multiply by 1.1

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>200</td>
</tr>
</tbody>
</table>
NewSQL

- Tables are replicated
- Enforce operation order across replicas

Deposit
Add 100

Interest
Multiply by 1.1

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>200</td>
</tr>
</tbody>
</table>
NewSQL

• Tables are replicated
• Enforce operation order across replicas

Deposit
Add 100

Interest
Multiply by 1.1

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>220</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>220</td>
</tr>
</tbody>
</table>
NewSQL

- Tables are replicated
- Enforce operation order across replicas

Deposit
Add 100

Interest
Multiply by 1.1

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>220</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>220</td>
</tr>
</tbody>
</table>
NewSQL

• **Single machine bottlenecks:**
  - Logging & locking

• **Multiple machine bottlenecks:**
  - Replication, locks, joins
NewSQL

• When to use VoltDB?
  – run at scale
  – You need 100% availability
  – You need ACID
Conclusion
Conclusion

• Hardware is cheaper
Conclusion

Transactional Database

Data warehouses

Row-store

Column-store
Conclusion

• Single machine bottlenecks:
  
• Multiple machines bottlenecks:
  
  Logging & locking

  Replication, locks, joins
Conclusion

- **NoSQL** adapts by simplifying
  - No ACID
  - No joins
- **NewSQL** adapts by reengineering
  - ACID
  - Removes concurrency
  - Simplifies logging
  - Smart but limited partitioning across servers
Conclusion

• **Disclaimer:** there is much more
  - Scientific databases: SciDB
  - Time series databases: InfluxDB
  - Graph databases: neo4j

• **Caveat:** rapid changes

• But hopefully now you have reasoning tools
Conclusion

• Thanks!