Introduction to Data Management

Wide Column Data Stores

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Recap: NoSQL in a Nutshell

- NoSQL → Looser data model
  - Give up built-in OLAP/analysis functionality
  - Give up built-in ACID consistency
Outline

- Wide-Column Store
  - "2D" Hash Table (Row → Column → Blob)
- NoSQL use case comparison
NoSQL Data Models

Key-Value Database

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA, BBB, CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA, BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA, DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA, 2,01/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3, ZZZ, 5623</td>
</tr>
</tbody>
</table>

Wide-Column Store (Extensible Record Store)

Graph Database

Document Store

XML

```xml
<employees>
  <employee>
    <name>James Kirk</name>
    <age>40</age>
  </employee>
  <employee>
    <name>Jean-Luc Picard</name>
    <age>45</age>
  </employee>
  <employee>
    <name>Wesley Crusher</name>
    <age>27</age>
  </employee>
</employees>
```

JSON

```json
{
  "employees": [
    {
      "name": "James Kirk",
      "age": 40,
    },
    {
      "name": "Jean-Luc Picard",
      "age": 45,
    },
    {
      "name": "Wesley Crusher",
      "age": 27,
    }
  ]
}
```
NoSQL Data Models

Key-Value Database

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA, BBB, CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA, BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA, DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA, 2,01/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3, ZZZ, 5623</td>
</tr>
</tbody>
</table>

Wide-Column Store (Extensible Record Store)

- Row + column key to value pairs
- “A multidimensional hash table”

Graph Database

Document Store

XML

```
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

JSON

```
{  
  "empinfo" :  
    {  
      "employees" :  
        [  
          {  
            "name" : "James Kirk",  
            "age" : 40,  
          },  
          {  
            "name" : "Jean-Luc Picard",  
            "age" : 45,  
          },  
          {  
            "name" : "Wesley Crusher",  
            "age" : 27,  
          }  
        ]  
    }  
}
```
NoSQL Data Models

### Key-Value Database

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA, BBB, CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA, BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA, DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA, 201/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3, ZZZ, 5623</td>
</tr>
</tbody>
</table>

### Wide-Column Store (Extensible Record Store)

- Google Bigtable
- Apache HBase
- Cassandra

### Graph Database

- User: Peter
- Follows: Johan, Emil
- User: Johan
- Follows: Emil

### Document Store

**XML**

```xml
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

**JSON**

```json
{   "empinfo": [
  {
    "employees": [
      {
        "name": "James Kirk",
        "age": 40,
      },
      {
        "name": "Jean-Luc Picard",
        "age": 45,
      },
      {
        "name": "Wesley Crusher",
        "age": 27,
      }
    ]
  }
]}
```
Known Wide-Column Store Applications

- **Google Search, Gmail, Docs, ...** (Bigtable)
- **Facebook Messages** (HBase → HydraBase)
- **Netflix Viewing History** (Cassandra)
- **Spotify Personalization** (Cassandra)
Wide-Column Stores

- Similar to relational models with differences
  - Has **notion of rows** (objects) and **columns** (attributes)
  - **Schemaless**: A row may have columns that are not necessarily the same as other rows
  - All rows must have a unique key
  - All values must be accompanied by timestamps
A **cell** holds a value and write timestamp

**Rows** contain **columns** that point to cells

**Column families** define groups of columns where each column has a **qualifier**

A **keyspace/table** is:
- A set of column families containing rows
- A set of rows containing column families
Data Model Basics

Keypoint

Column Family

Row Key

Column

Value

Timestamp

Column

Value

Timestamp

Row Key

Column

Value

Timestamp

Column

Value

Timestamp

Column

Value

Timestamp
Data Model Basics

Data Access Path

Row Key → Column → Cell
Data Model Basics

Data Access Path

Row Key → Column → Cell

Row Key → Col. Fam. + Qualifier → Value Timestamp
Data Model Basics

Data Access Path

Row Key → Column → Cell

Row Key → Col. Fam. + Qualifier → Value Timestamp

“com.google.www” → “content:html” → “<html>…” 1557745425
Data Types

- Mostly schemaless, untyped data
  - Store byte data instead
- Native support depends on implementation:
  - Bigtable/HBase 🡪 byte arrays only
  - Cassandra 🡪 int, text, blob, …
Literally No SQL

- Most do not natively support SQL-like languages
  - Schemaless data? Bulk referencing columns doesn’t make sense
  - Byte-only data types? Query manipulations useless

- Queries are normally done through API interface
Simplified API Overview

- Put
- Get/Result
- Scan/ResultScanner
- Filter (Abstract Class)
  - MultiRowRangeFilter/RowRange
  - ColumnPrefixFilter
  - SingleColumnValueFilter/CompareOp
Physical Considerations

- Individual rows are never split across nodes
- **Rows are stored in (lexicographic) order**
  - Row key sorting is typically the only form of index in the entire database
Physical Considerations

Row Key  Data
1524...  (col/value)
...
1525...  (col/value)
1526...  (col/value)
...
1527...  (col/value)
1528...  (col/value)
...
1529...  (col/value)

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Physical Considerations

Row Key □ Data
1524... □ (col/value)
...
1525... □ (col/value)
1526... □ (col/value)
...
1527... □ (col/value)
1528... □ (col/value)
...
1529... □ (col/value)

Node 1
1524... □ (col/value)
...
1525... □ (col/value)

Node 2
1526... □ (col/value)
...
1527... □ (col/value)

Node 3
1528... □ (col/value)
...
1529... □ (col/value)
Physical Considerations

Row Key → Data
1524… (col/value)
...
1525… (col/value)
1526… (col/value)
...
1527… (col/value)
1528… (col/value)
...
1529… (col/value)

Index

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1524…</td>
<td></td>
</tr>
<tr>
<td>1526…</td>
<td></td>
</tr>
<tr>
<td>1528…</td>
<td></td>
</tr>
</tbody>
</table>

Node 1
1524… (col/value)
...
1525… (col/value)

Node 2
1526… (col/value)
...
1527… (col/value)

Node 3
1528… (col/value)
...
1529… (col/value)
Row Keys

- Row key choice is flexible
  - Auto-generated or provided IDs are possible
- Compound keys are often used
  - String concatenation is easy to manage
  - First component of key is called the **key prefix**
  - Secondary components are **tags**
Row Keys

- Row key choice is flexible
  - Auto-generated or provided IDs are possible

- Compound keys are often used
  - String concatenation is easy to manage
  - First component of key is called the **key prefix**
  - Secondary components are **tags**

Example:
A combination of timestamp and user ID uniquely identifies row.
Say my workload would benefit from sorting on a user ID.
Row Keys

- Row key choice is flexible
  - Auto-generated or provided IDs are possible

- Compound keys are often used
  - String concatenation is easy to manage
  - First component of key is called the **key prefix**
  - Secondary components are **tags**

Example:
A combination of timestamp and user ID uniquely identifies row.
Say my workload would benefit from sorting on a user ID

Timestamp: 1557745425  
UserID: “noobmaster69”
Row Keys

- Row key choice is flexible
  - Auto-generated or provided IDs are possible

- Compound keys are often used
  - String concatenation is easy to manage
  - First component of key is called the **key prefix**
  - Secondary components are **tags**

Example:
A combination of timestamp and user ID uniquely identifies row. Say my workload would benefit from sorting on a user ID

Timestamp: 1557745425  
UserID: “noobmaster69”

Row Key: “noobmaster69#1557745425”
Row Keys

- Row key choice is flexible
  - Auto-generated or provided IDs are possible
- Compound keys are often used
  - String concatenation is easy to manage
  - First component of key is called the **key prefix**
  - Secondary components are **tags**

Example:
A combination of timestamp and user ID uniquely identifies row. Say my workload would benefit from sorting on a user ID

Timestamp: 1557745425  
UserID: “noobmaster69”

Row Key: “noobmaster69#1557745425”

Key prefix
Row Keys

- Row key choice is flexible
  - Auto-generated or provided IDs are possible

- Compound keys are often used
  - String concatenation is easy to manage
  - First component of key is called the **key prefix**
  - Secondary components are **tags**

Example:
A combination of timestamp and user ID uniquely identifies row. Say my workload would benefit from sorting on a user ID

Timestamp: 1557745425  
UserID: “noobmaster69”

Row Key: “**noobmaster69**#1557745425”

  - Key prefix
  - Tag
Database Performance

- One critical aspect of wide-column store performance is **massive parallelization**
- Choice of key prefix impacts:
  - Hotspotting avoidance
  - Fast range queries
Time-Series Data

- Common use case for wide-column store
- Financial trading data (HBase → OpenTSDB)
  - Stocks prices update on every trade
  - High-frequency trading needs millisecond responses
- IoT sensor data
  - Sampling rates could produce GBs of data in a minute
  - Humans are able to notice sub-second responsiveness
Hotspotting

- **Avoid monotonically increasing row keys**
  - Time series data
  - Counters

- **Fix hotspotting via:**
  - **Field promotion**
  - **Salting**
Hotspotting

1524… □ (user/"noobmaster69")
...
1525… □ (user/"420_E-Sports_Masta")
1526… □ (user/"[720NoScope]Headshotz")
...
1527… □ (user/"bobtheninja246")
1528… □ (user/"bobtheninja246")
...
1529… □ (user/"420_E-Sports_Masta")

Field promotion

“noobmaster69#1524…” □ (user/"noobmaster69")
...
“420_E-Sports_Masta#1525…” □ (user/"420_E-Sports_Masta")
“[720NoScope]Headshotz#1526…” □ (user/"[720NoScope]Headshotz")
...
“bobtheninja246#1527…” □ (user/"bobtheninja246")
“bobtheninja246#1528…” □ (user/"bobtheninja246")
...
“420_E-Sports_Masta#1529…” □ (user/"420_E-Sports_Masta")
Hotspotting

1524... □ (user/”noobmaster69”)
...  
1525... □ (user/”420_E-Sports_Masta”)
1526... □ (user/”[720NoScope]Headshotz”)
...  
1527... □ (user/”bobtheninja246”)
1528... □ (user/”bobtheninja246”)
...  
1529... □ (user/”420_E-Sports_Masta”)

Salting

“EWIC#1524...” □ (user/”noobmaster69”)
...  
“41Z3#1525...” □ (user/”420_E-Sports_Masta”)
“A91I#1526...” □ (user/”[720NoScope]Headshotz”)
...  
“8M30#1527...” □ (user/”bobtheninja246”)
“69MC#1528...” □ (user/”bobtheninja246”)
...  
“0O92#1529...” □ (user/”420_E-Sports_Masta”)

March 11, 2020

NoSQL
Faster Range Queries

- Like clustered index in RDBMS
- Pick the key prefix that will make the most sense given a workload
Row Key Tricks

• How should we construct the database row key for an internet content database?

Example:
Queries want to look at data within the same domain.
Row Key Tricks

- How should we construct the database row key for an internet content database?

Example:
Queries want to look at data within the same domain.

Domain
maps.apple.com
maps.google.com
docs.google.com
cloud.google.com
cloud.ibm.com
cloud.oracle.com
Row Key Tricks

- How should we construct the database row key for an internet content database?

Example:
Queries want to look at data within the same domain.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Reverse Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>maps.apple.com</td>
<td>com.apple.maps</td>
</tr>
<tr>
<td>maps.google.com</td>
<td>com.google.cloud</td>
</tr>
<tr>
<td>docs.google.com</td>
<td>com.google.docs</td>
</tr>
<tr>
<td>cloud.google.com</td>
<td>com.google.maps</td>
</tr>
<tr>
<td>cloud.ibm.com</td>
<td>com.ibm.cloud</td>
</tr>
<tr>
<td>cloud.oracle.com</td>
<td>com.oracle.cloud</td>
</tr>
</tbody>
</table>
Choosing Wide-Column Stores

- NoSQL vs Relational tradeoffs apply
  - Schema vs schemaless
  - Consistency vs availability
  - Scalability and usability
  - ...

- Sparse data
  - Ex: Matrix representation of a graph

- Evolving schema
  - Schemaless structure easily allows new attributes

- Analytical applications
  - Note: application does analysis not database
NoSQL on the Scale Up Problem

- **KV Store**
  - Hash Table (Key ✡ Blob)

- **Wide-Column Store**
  - "2D" Hash Table (Row ✡ Column ✡ Blob)

- **Document Store**
  - Hash Table + Parsable Documents

Trade off: well-defined data vs speed
No database paradigm is "better" than another

One-size does not fit all (M. Stonebraker)
Key-Value Stores

- **Pros:**
  - Fast lookups/writes of single entities
  - Simple API
    - Schemaless structure easily allows new attributes
- **Cons:**
  - Expensive/impossible to do range queries
  - Can’t operate over multiple keys at once
  - Application must deserialize the values
  - Need offline-processing (think: MapReduce) to do analytics
  - No ACID
Wide-Column Stores

- **Pros:**
  - **Good for sparse data**
  - **Good for evolving schema**
    - Schemaless structure easily allows new attributes
  - **Analytical applications possible**
    - Can efficiently do range queries, but application does analysis not database

- **Cons:**
  - **Lookup only possible on the keys**
  - **No ACID**
Document Stores

▪ Pros:
  • Fast lookups/writes of single entities
  • Semi-structured
    • Still able to change schema, and now the db can do more for us
  • Multiple types of lookup possible
    • Can create indexes to speed up other types of queries

▪ Cons:
  • Less structured than RDBMS
  • Distributed system still harder to manage than single node
  • No ACID
Glorified Hash Tables

- **KV Store**
  - Hash Table (Key 🡪 Blob)

- **Wide-Column Store**
  - "2D" Hash Table (Row 🡪 Column 🡪 Blob)

- **Document Store**
  - Hash Table + Parsable Documents

Trade off: well-defined data vs speed
Graph Databases

- **Pros:**
  - **Fast lookups of connected data**
    - Unlike RDBMS, no need for expensive joins (think: 10’s of joins)
    - Unlike other NoSQL, models connected data
  - **ACID guarantees**

- **Cons:**
  - **Optimized for reading heavily-connected data**
    - Write-heavy workloads won’t benefit
  - **Lack of standardization**
NoSQL vs SQL

**NoSQL:**
- Simple API but fewer features
- Loose schema
  - Easy to change, but application does more work
- BASE guarantees harder for developers to understand
- Administration of distributed system hard

**SQL:**
- Express nearly any type of query
- Strict schema
  - Harder to change, but db built-ins make queries easy
- ACID guarantees expensive to enforce
- Scaling on single node is hard
NewSQL addresses the issue of scalable relational data in high-profile data use cases:
- Banking/Billing
- User authorization
- Order fulfilment

First NewSQL system is H-Store in 2007 (commercialized as VoltDB)
Some NewSQL systems are like RDBMS on top of NoSQL systems

- Google implements Spanner on top of BigTable
- Cockroach Labs implements CockroachDB on top of RocksDB
• NewSQL vs traditional RDBMS: You only need a system that is as powerful as you need. Don’t take distributed overhead if you don’t need it.

• NewSQL is NoSQL: Relational models are still a poor choice when you have unstructured data.
NewSQL Features

- Simple to manage system
- Relational Transaction processing
- High availability
- Allows unstructured data

Traditional DBMS  NewSQL  NoSQL
How to choose?

▪ Research
▪ Benchmark: Make a mock dataset and workload and try it out
Keep Multiple Copies

- For analytics: Sync data to a “reporting database”
- For caching: Sync data to a KV store
- For speeding up different queries: Keep two copies sorted differently
Multi Model Databases

- KV Store vs RDBMS vs Graph DB: why not both?
- Some DBMSs bake in multiple models and optimize them together as well as they can.
Consider the human side

- Ramp up costs for learning a new system
- Costs of mistakes when you do something wrong
- Maintenance costs for maintaining multiple systems
Questions to ask yourself

▪ How large is my data going to be, realistically?
▪ Am I just looking for an excuse to use the latest technology?
▪ What tools do I and my teammates know how to use best?
▪ What is my query load going to look like?
▪ Will I need to change my schema later?
▪ Do I need ACID guarantees?
▪ ...

March 11, 2020

NoSQL
In short...

- There's no such thing as a free lunch