Introduction to Data Management

NoSQL: Key Value and Wide Column

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
Recap: Let’s Drop ACID

- RDBMSs have the ACID consistency model
- NoSQL sys. have the **BASE** consistency model

**Basically Available**
- Most failures do not cause a complete system outage

**Soft state**
- System is not always write-consistent

**Eventually consistent**
- Data will eventually converge to agreed values
Recap: CAP Theorem

- Old name: Brewer’s Conjecture
- In a distributed data store, one can only provide two of the following three guarantees:
  - **Consistency**
    - Every read receives the most recent write or an error
  - **Availability**
    - Every request must respond with a non-error
  - **Partition tolerance**
    - Continued operation in presence of dropped or delayed messages
Outline

- KV Store
  - Hash Table (Key → Blob)

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

- Document Store
  - Hash Table + Parsable Documents
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)

**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,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
<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,
      }
    ]
  }
}
```
NoSQL Data Models

Key-Value Database

- Key to value pairs
- “A hash table”

Wide-Column Store
(Extensible Record Store)

Graph Database

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
      }
    ]
  }
}
```
NoSQL Data Models

### Key-Value Database

- Amazon DynamoDB
- RocksDB
- Redis

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

#### XML

```xml
<eminfo>
  <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>
</eminfo>
```

#### JSON

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

### Graph Database

- User: Peter
- User: Johan
- User: Emil

### Document Store

- User
  - name: Peter
  - age: 40
  - email: peter@example.com
- User: Johan
  - name: Johan
  - age: 45
  - email: johan@example.com
- User: Emil
  - name: Emil
  - age: 27
  - email: emil@example.com
NoSQL Data Models

**Key-Value Database**
- Amazon DynamoDB
- RocksDB
- Redis

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

**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**
  - Amazon DynamoDB
  - RocksDB
  - Redis

- **Graph Database**
  - Extended to also be a Document Store

- **Document Store**
  - XML
  ```xml
  <employee>
    <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>
  </employee>
  ```
  - JSON
  ```json
  { "employee" : [
    { "employees" : [
      { "name" : "James Kirk", "age" : 40, }]
    },
    { "name" : "Jean-Luc Picard", "age" : 45, },
    { "name" : "Wesley Crusher", "age" : 27, }
  ] }```

- **In-memory KV store**
  - In-memory KV store

- **Persistent KV store**
  - Persistent KV store
Key-Value Store

- Data model:
  - (key, value) pairs
  - Key – string/integer/…, unique for the entire data
  - Value – anything
Key-Value Store

- Data model:
  - (key, value) pairs
  - Key 🡪 string/integer/..., unique for the entire data
  - Value 🡪 anything

- Basic Operations:
  - get(key)
  - put(key, value)
Key-Value Store

- Data model:
  - (key, value) pairs
  - Key \( \rightarrow \) string/integer/…, unique for the entire data
  - Value \( \rightarrow \) anything

- Basic Operations:
  - get(key)
  - put(key, value)

- Distribution/Partitioning:
  - Access via hash function
  - No replication: Key k stored at server \( h(k) \mod N \)
  - 3-way replication: Key k stored at servers \( h_1(k) \mod N, h_2(k) \mod N, h_3(k) \mod N \)
Key-Value Modeling

Represent all Flights as KV pairs

Potential KV pairings

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
</table>

NoSQL
Key-Value Modeling

Represent all Flights as KV pairs

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FID</td>
<td>Single flight record</td>
</tr>
</tbody>
</table>
Key-Value Modeling

Represent all Flights as KV pairs

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FID</td>
<td>Single flight record</td>
</tr>
<tr>
<td>Date</td>
<td>All flight records on that day</td>
</tr>
</tbody>
</table>
Key-Value Modeling

Represent all Flights as KV pairs

Potential KV pairings

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FID</td>
<td>Single flight record</td>
</tr>
<tr>
<td>Date</td>
<td>All flight records on that day</td>
</tr>
<tr>
<td>(origin, destination)</td>
<td>All flight records between the cities</td>
</tr>
</tbody>
</table>
DynamoDB API

- Create, Read, Update, Delete (CRUD) actions
  - Create ▸ **PutItem**
  - Read ▸ **GetItem**
  - Update ▸ **UpdateItem** (Document store functionality)
  - Delete ▸ **DeleteItem**

- Read consistency
  - Eventually consistent (default, may be stale data)
  - Strongly consistent (gets most recent written data)

- As of December 2018, ACID is “supported”
  - **TransactWriteItems**
  - **TransactGetItems**
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

![Graph Database Diagram]

### 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
      }
    ]
  }
}
```
NoSQL Data Models

### Key-Value Database

- **Table:**
  - **Key:** K1, K2, K3, K4, K5
  - **Value:** AAA, BBB, CCC, AAA, BBB, AAA, DDD, AAA, 201/01/2015, 3, ZZZ, 5623

### Wide-Column Store

*(Extensible Record Store)*

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

### Graph Database

- **Diagram:** Relationships between entities

### 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,2,01/01/2015</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**

**Document Store**

**XML**

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

**JSON**

```json
{  
  "eminfo" :  
    {  
      "employees" : [  
        {  
          "name" : "James Kirk",  
          "age" : 40,  
        },  
        {  
          "name" : "Jean-Luc Picard",  
          "age" : 45,  
        }
      ]
    }
}
```
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**
Data Model Basics

- **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

Keyspace

Column Family

Row Key

Column
Value
Timestamp

Column
Value
Timestamp

Row Key

Column
Value
Timestamp

Column
Value
Timestamp

Column
Value
Timestamp
Data Model Basics

Row Key → Column → Cell

Data Access Path
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, ...
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)
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>1525...</td>
<td></td>
</tr>
<tr>
<td>1526...</td>
<td></td>
</tr>
<tr>
<td>1527...</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 a 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
- **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")
...
“4I3Z#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")
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.
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
On A Practical Note

- No database paradigm is "better" than another
- One-size does not fit all (M. Stonebraker)