NoSQL Databases

Polytech Nice-Sophia, SI4

Pierre Monnin

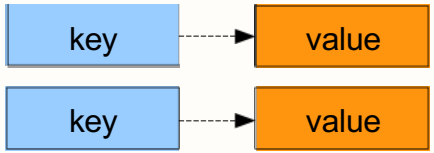
pierre.monnin@inria.fr

Special thanks to **Dr. Sabeur Aridhi**, **Dr. Jörn Franke** and **Dr. Nicolas Travers** for providing their materials

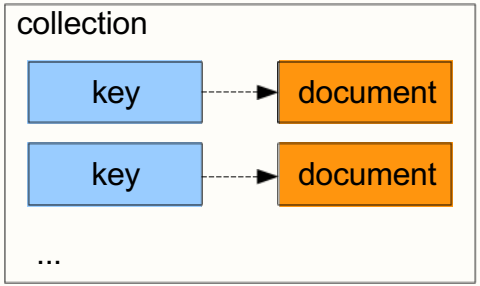
Chapter 4: Column-Oriented Databases

Types of data stores

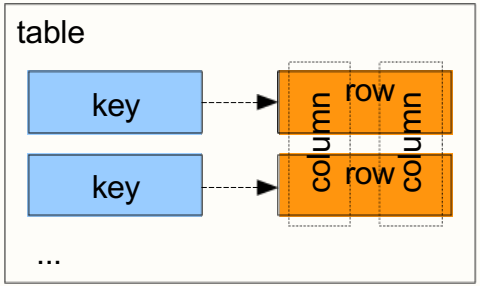
Key / value stores (opaque / typed)



Document stores (non-shaped / shaped)



Relational databases

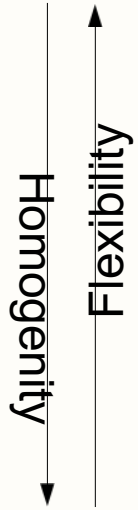


Data stores, summary

Documents in **document stores** can be heterogenous:
Different documents can have different attributes and types
Type information is stored per document or group of documents

Values in **key / value stores** can be heterogenous:
Different values can have different types
Type information is either not stored at all or stored per value

Rows in **relational databases** are homogenous:
Different rows have the same columns and types
Type information is stored once per column



Which data store type is ideal for the job depends on
the types of queries that must be run on the data!

Row vs. columnar relational databases

All relational databases deal with tables, rows, and columns

But there are sub-types:

- row-oriented: they are internally organised around the handling of rows
- columnar / column-oriented: these mainly work with columns

Both types usually offer SQL interfaces and produce tables (with rows and columns) as their result sets

Both types can generally solve the same queries

Both types have specific use cases that they're good for (and use cases that they're not good for)

Row vs. columnar relational databases

In practice, row-oriented databases are often optimised and particularly good for OLTP workloads
*(**o**n**l**ine **t**ransaction **p**rocessing)*

whereas column-oriented databases are often well-suited for OLAP workloads
(***o**n**l**ine **a**nalytical **p**rocessing*)

this is due to the different internal designs of row- and column-oriented databases

Row-oriented storage

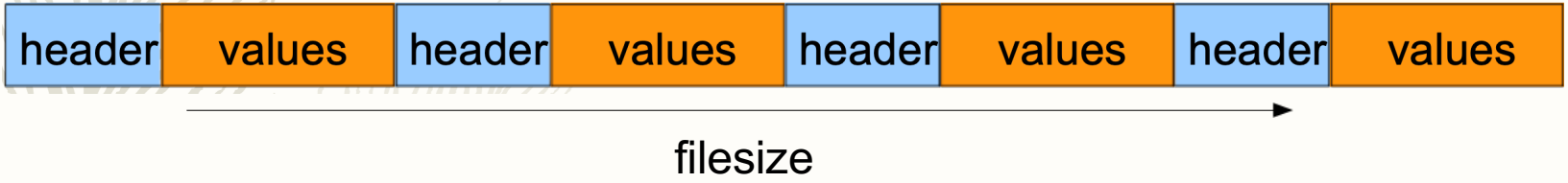
In row-oriented databases, row value data is usually stored contiguously:

row0 header	column0 value	column1 value	column2 value	column3 value
row1 header	column0 value	column1 value	column2 value	column3 value
row2 header	column0 value	column1 value	column2 value	column3 value

(the row headers contain record lengths, NULL bits etc.)

Row-oriented storage

When looking at a table's datafile, it could look like this:



Actual row values are stored at specific offsets of the values struct:

Offsets depend on column types, e.g. 4 for int32, 8 for int64 etc.

Row-oriented storage

To read a specific row, we need to determine its position first

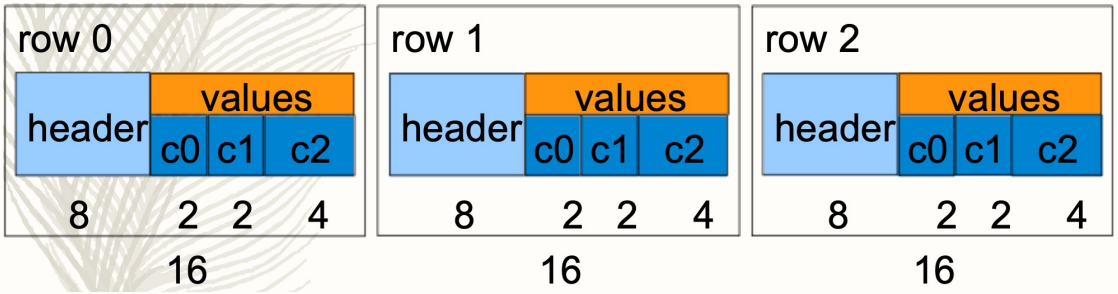
This is easy if rows have a fixed length:

position = row number * row length + header length

header length = 8

row length = header length (8) + value length (8) = 16

value length = c0 length (2) + c1 length (2) + c2 length (4) = 8



Row-oriented storage

In reality its often not that easy

With varchars, values can have variable lengths

With variable length values, we cannot simply jump in the middle of the datafile because we don't know where a row starts

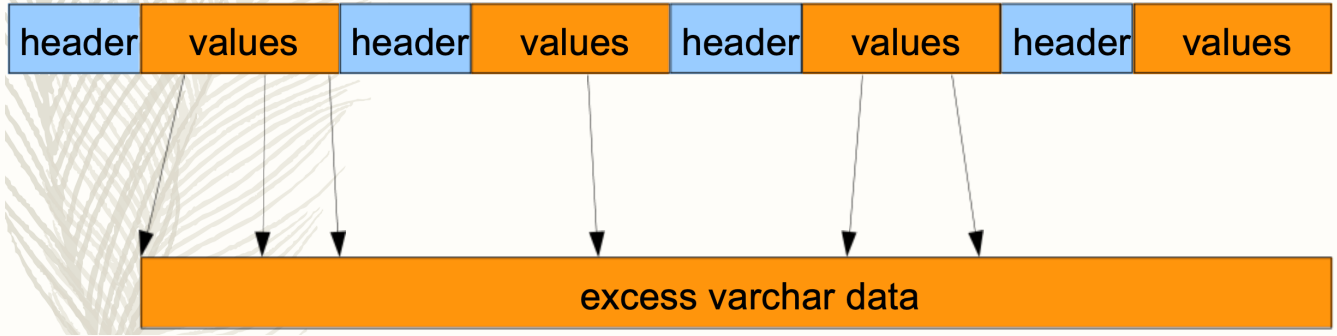
To calculate the offset of a specifc row, we need to use the record lengths stored in the headers. This means traversing headers!

Row-oriented storage

Another strategy is to split varchar values into two parts:

- prefixes of fixed-length are stored in-row
- excess data exceeding the prefix length are stored off-row, and are pointed to from the row

This allows values to have fixed lengths again, at the price of additional lookups for the excess data:

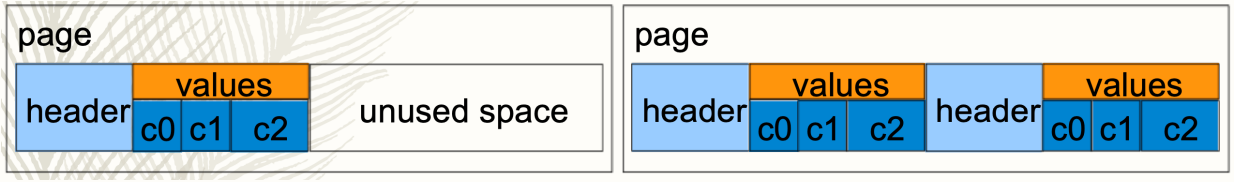


Row-oriented storage

Datafiles are often organised in pages

To read data for a specific row, the full page needs to be read

The smaller the rows, the more will fit onto a page



In reality, pages are often not fully filled up entirely

This allows later update-in-place without page splits or movements (these are expensive operations) but may waste a lot of space

Furthermore, rows must fit onto pages, leaving parts unused

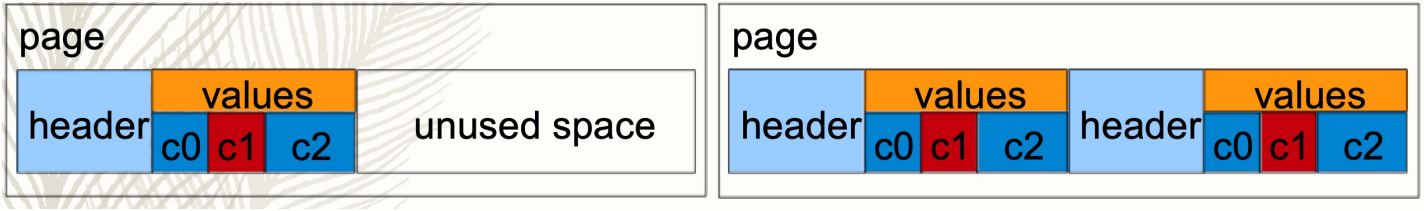
Row-oriented storage

- Row-oriented storage is good if we need to touch one row. This normally requires reading/writing a single page
- Row-oriented storage is beneficial if all or most columns of a row need to be read or written. This can be done with a single read/write.
- Row-oriented storage is very inefficient if not all columns are needed but a lot of rows need to be read:
  - Full rows are read, including columns not used by a query
  - Reads are done page-wise. Not many rows may fit on a page when rows are big
  - Pages are normally not fully filled, which leads to reading lots of unused areas
  - Record (and sometimes page) headers need to be read, too but do not contain actual row data

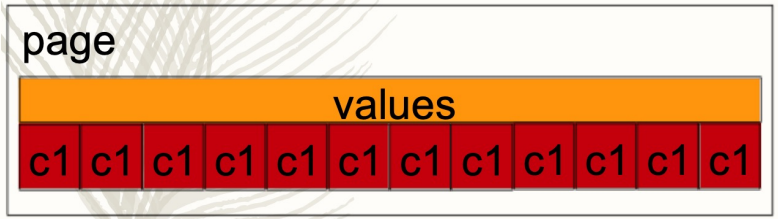
Row-oriented storage

Row-oriented storage is especially inefficient when only a small amout of columns is needed but the table has many columns

Example: need only c1, but must read 2 pages to get just 3 values!



We'd prefer getting 12 values with 1 read. This would be optimal:



Column-oriented storage

- Column-oriented databases primarily work on columns
- All columns are treated individually
- Values of a single column are stored contiguously
- This allows array-processing the values of a column
- Rows may be constructed from column values later if required
- This means column stores can still produce row output (tables)
- Values from multiple columns need to be retrieved and assembled for that, making implementation of bit more complex
- Query processors in columnar databases work on columns, too

Column-oriented storage

Column stores can greatly improve the performance of queries that only touch a small amount of columns

This is because they will only access these columns' particular data

Simple math: table t has a total of 10 GB data, with

- column a: 4 GB
- column b: 2 GB
- column c: 3 GB
- column d: 1 GB

If a query only uses column d, at most 1 GB of data will be processed by a column store

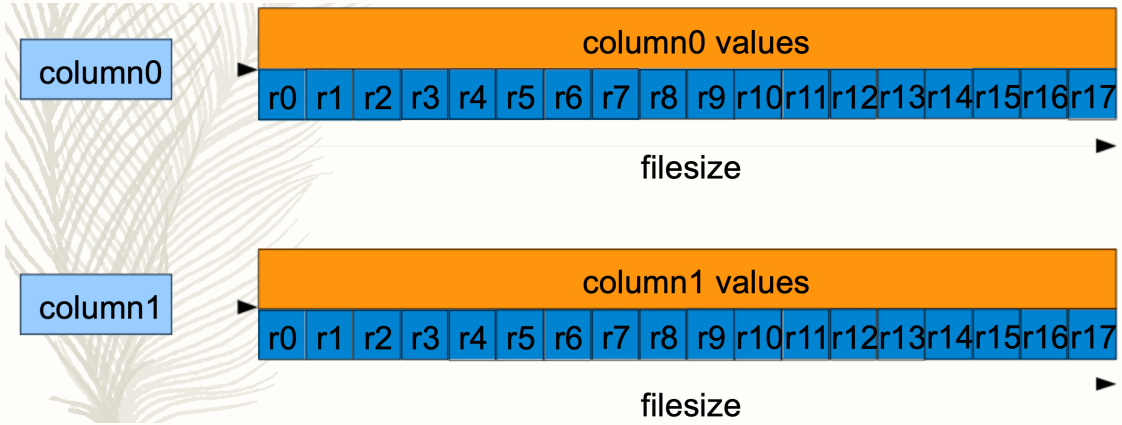
In a row store, the full 10 GB will be processed

Column-oriented storage

Column stores store data in column-specific files

Simplest case: one datafile per column

Row values for each column are stored contiguously



Column-oriented storage

- With every page read, we can read the values for many rows at once. We'll get more values per I/O than with row-storage
- This is ideal if we want to process most records anyway
- If values are fixed-length, we are also allowed random access to each record's value
- Saving header information in the values files is undesired.

  Can save meta information separately, e.g. NULL bits as a bit vector per column

Column-oriented storage

- All data within each column datafile have the same type, making it ideal for compression
- Usually a much better compression factor can be achieved for single columns than for entire rows
- Compression allows reducing disk I/O when reading/writing column data but has some CPU cost
- For data sets bigger than the memory size, compression is often beneficial because disk access is slower than decompression

Column-oriented storage

- A good use case for compression in column stores is dictionary compression for variable length string values
- Each unique string is assigned an integer number
- The dictionary, consisting of integer number and string value, is saved as column meta data
- Column values are then integers only, making them small and fixed width
- This can save much space if string values are non-unique
- With dictionaries sorted by column value, this will also allow rangequeries

Column-oriented storage

- In a column database, it is relatively cheap to traverse all values of a column
- Building an index on a column only requires reading that column's data, not the complete table data as in a row store
- Adding or deleting columns is a relatively cheap operation, too

Column-oriented storage, segments

- Column data in column stores is ofted grouped into segments/packets of a specific size (e.g. 64 K values)
- Meta data is calculated and stored separately per segment, e.g.:

  min value in segment

  max value in segment

  number of NOT NULL values in segment

  histograms

  compression meta data

Column-oriented storage, segments

- Segment meta data can be checked during query processing when no indexes are available
- Segment meta data may provide information about whether the segment can be skipped entirely, allowing to reduce the number of values that need to be processed in the query
- Calculating segment meta data is a relatively cheap operation (only needs to traverse column values in segment) but still should occur infrequently
- In a read-only or read-mostly workload like this is tolerable

Column-oriented processing

- Column values are not processed row-at-a-time, but block-at-a-time
- This reduces the number of function calls (function call per block of values, but not per row)
- Operating in blocks allows compiler optimisations, e.g. loop unrolling, parallelisation, pipelining
- Column values are normally positioned in contiguous memory locations, also allowing SIMD operations (vectorisation)
- Working on many subsequent memory positions also improves cache usage (multiple values are in the same cache line) and reduces pipeline stalls
- All of the above make column stores ideal for batch processing

Column-oriented processing

- Reading all columns of a row is an expensive operation in a column store, so full row tuple construction is avoided or delayed as much as possible internally
- Updating or inserting rows may also be very expensive and may cost much more time than in a row store
- Some column stores are hybrids, with read-optimised (column) storage and write-optimised OLTP storage
- Still, column stores are not really made for OLTP workloads, and if you need to work with many columns at once, you'll pay a price in a column store

HBase A Comprehensive Introduction

Overview

Overview: History

- Began as project by Powerset to **process massive amounts of data** for natural language search

- Open-source implementation of Google's **BigTable**
  - Lots of **semi-structured data**
  - Commodity Hardware
  - Horizontal Scalability
  - Tight integration with **MapReduce**

- Developed as part of Apache's **Hadoop** project and runs on top of **HDFS (Hadoop Distributed Filesystem)**
  - Provides **fault-tolerant** way of storing **large quantities of sparse data**.

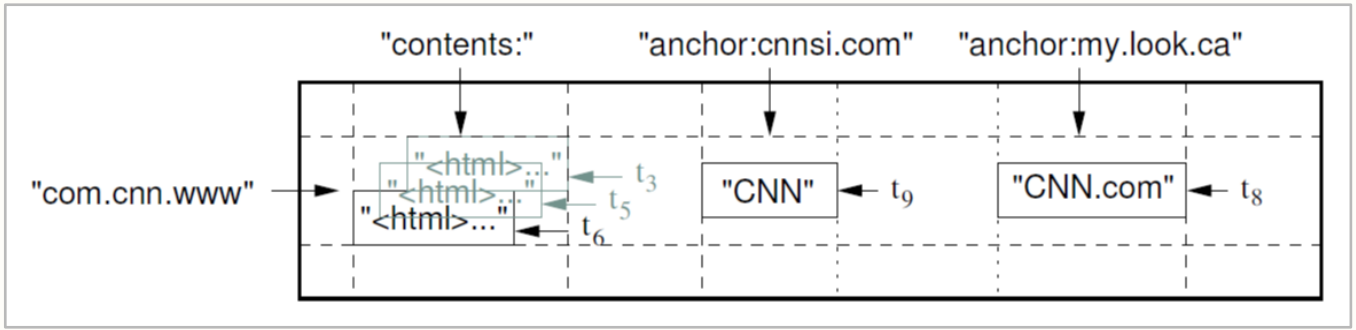
Overview: What is HBase?

- Non-relational, distributed database
- Column-Oriented
- Multi-Dimensional
- High Availability
- High Performance

Data Model & Operators

Data Model

- A **sparse**, **multi-dimensional**, **sorted** map
  - {row, column, timestamp} -> cell
- Column = **Column Family** : Column Qualifier



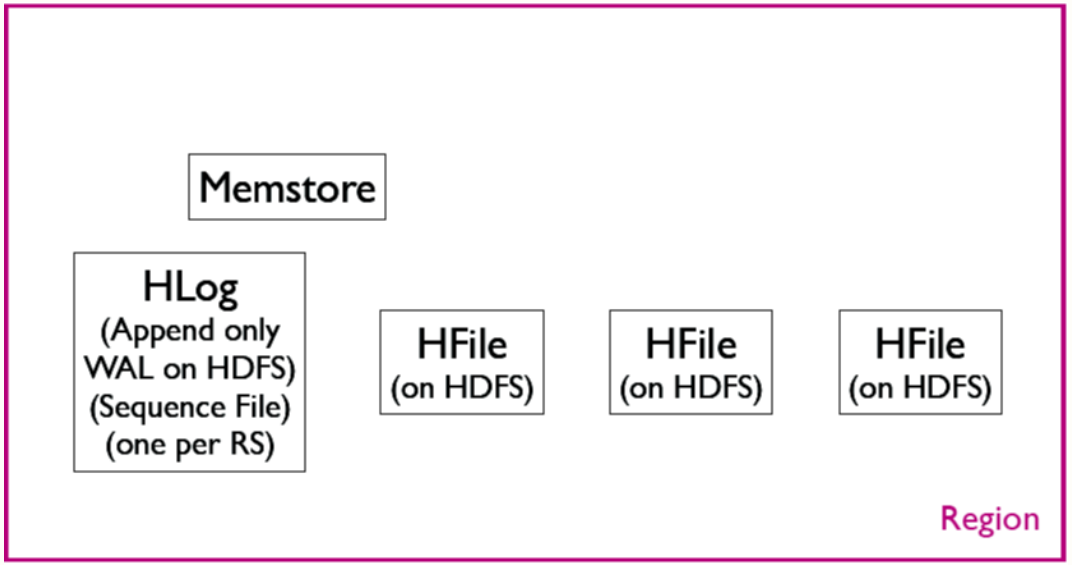
- Rows are **sorted lexicographically** based on row key
- **Region:** contiguous set of sorted rows
- HBase: a large number of columns, a low number of column families (2-3)

Operators

- Operations are based on **row keys**
- **Single-row operations:**
  - Put
  - Get
  - Scan
- **Multi-row operations:**
  - Scan
  - MultiPut
- No built-in joins (use MapReduce)

Physical Structures

Physical Structures: Data Organization



- **Region:** unit of distribution and availability
- Regions are split when grown too large
- Max region size is a tuning parameter
  - Too low: prevents parallel scalability
  - Too high: makes things slow

Physical Structures: Need for Indexes

- HBase has **no built-in support for secondary indexes**
- API only exposes operations by **row key**

Row Key	Name	Position	Nationality
"1"	Nowitzki, Dirk	PF	Germany
"2"	Kaman, Chris	C	Germany
"3"	Gasol, Paul	PF	Spain
"4"	Fernandez, Rudy	SG	Spain

- **Find all players from Spain?**
  - With built-in API, scan the entire table
  - Manually build a secondary index table
  - Exploit the fact that rows are sorted lexicographically by row key based on byte order

Physical Structures: Secondary Index

- Data Table:

Row Key	Name	Position	Nationality
“1”	Nowitzki, Dirk	PF	Germany
“2”	Kaman, Chris	C	Germany
“3”	Gasol, Paul	PF	Spain
“4”	Fernandez, Rudy	SG	Spain

- **Index table on nationality column**
  - a scan operation
  - start row = "Spain"
  - stop scanning: set a RowFilter with a BinaryPrefixComparator on the end value("Spain")
  - range queries are also supported

Row Key	Dummy
“Germany 1”	Germany 1
“Germany 2”	Germany 2
“Spain 3”	Spain 3
“Spain 4”	Spain 4

Physical Structures: Secondary Index (cont.)

- **Find all power forwards from Spain?**
  - A composite index

- Row keys are **plain byte arrays**
  - Byte order = your desired order?
  - Convert strings, integers, floats, decimals carefully to bytes
  - Default sorting is ascending; if descending indexes are needed, reverse bit order

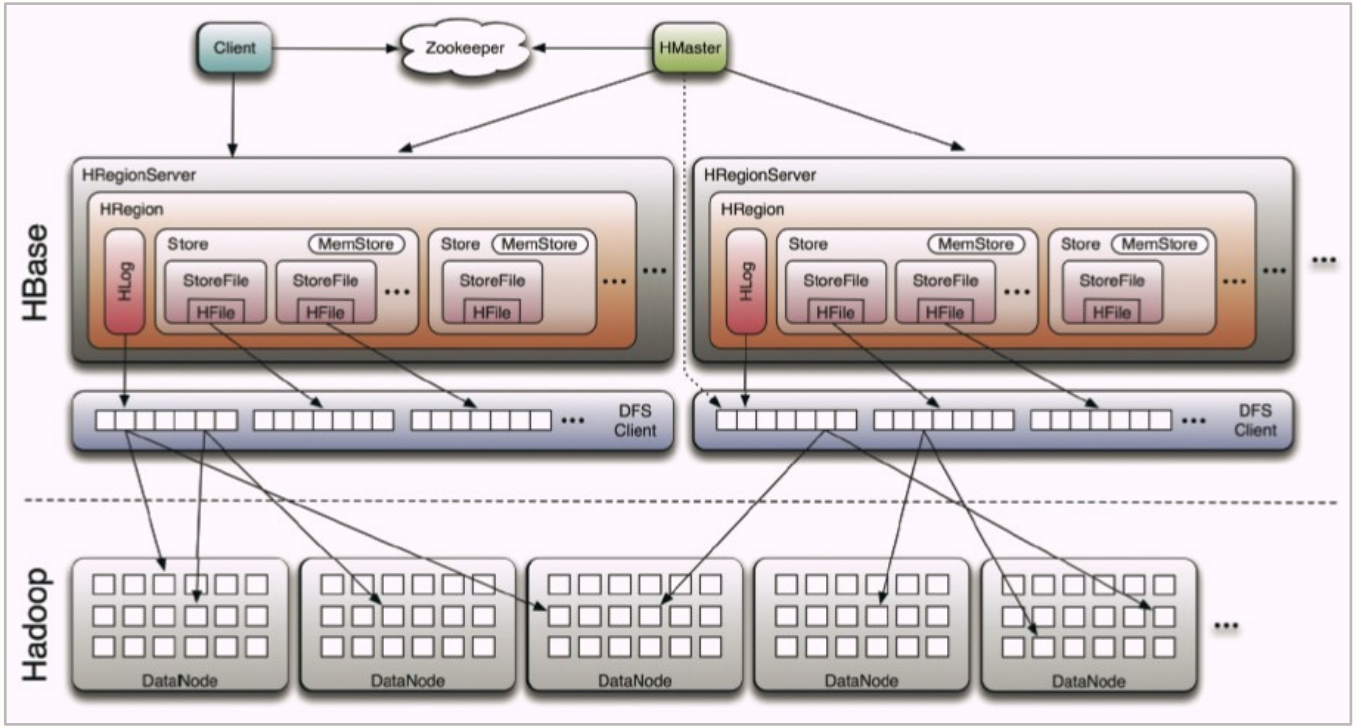
Physical Structures: More Indexing

- **Lily's** HBase Indexing Library
  - Aids in building and querying indexes in HBase
  - Hides the details of playing with byte[] row keys

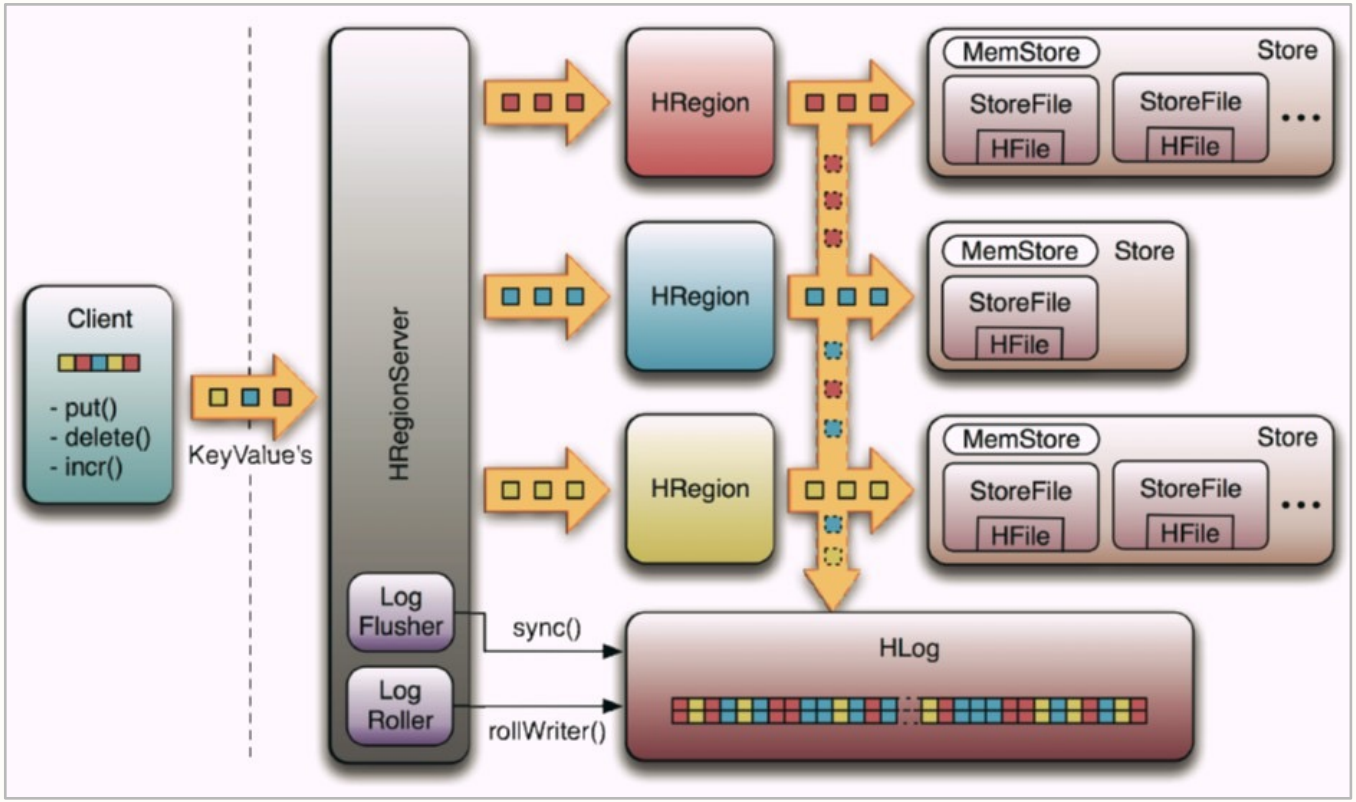
- **HBase + full text indexing and searching systems**
  - Apache Lucene (Apache Solr, elasticsearch)
  - Lily, HAvroBase (HBase + Solr), HBasene (HBase + Lucene)

System Architecture

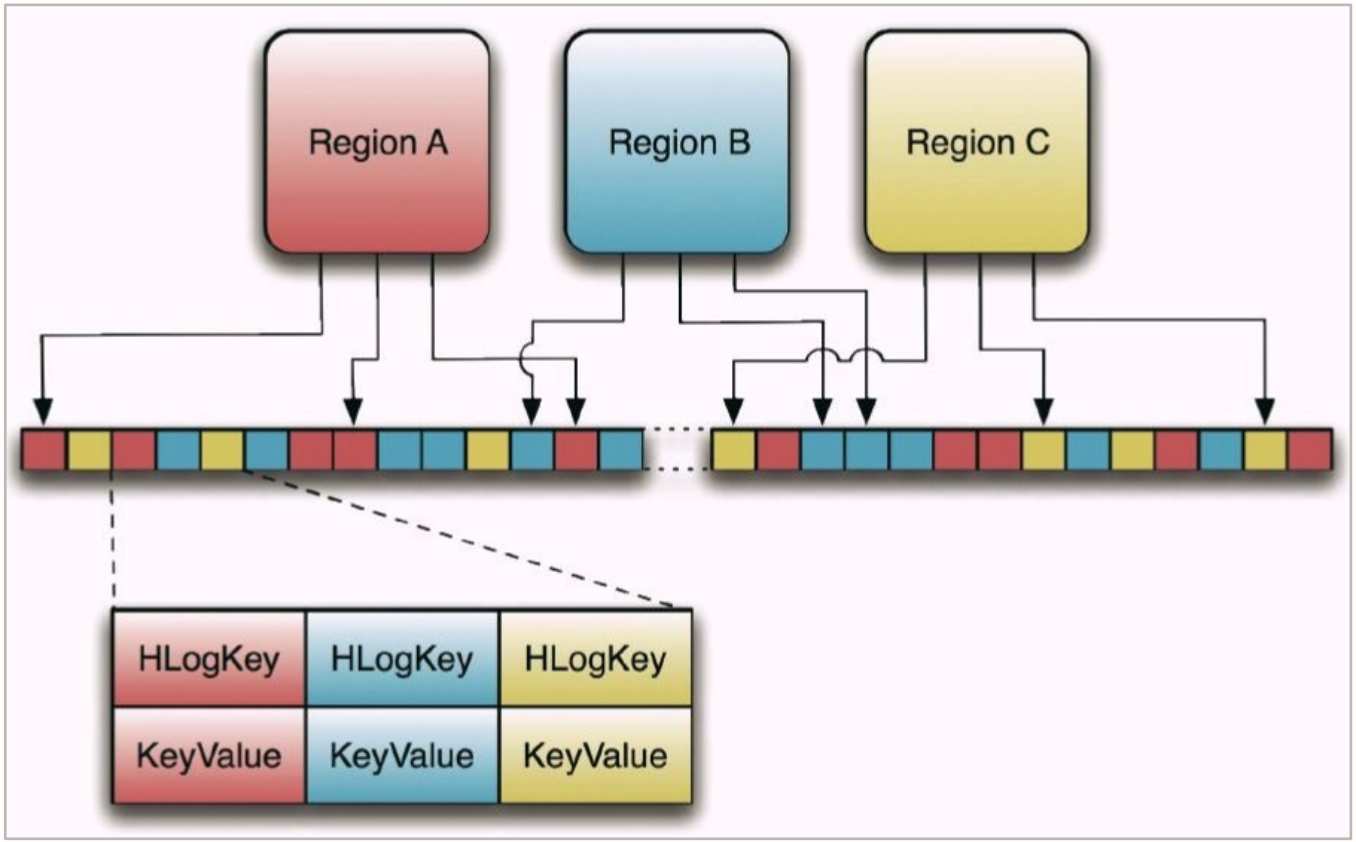
System Architecture: Overview



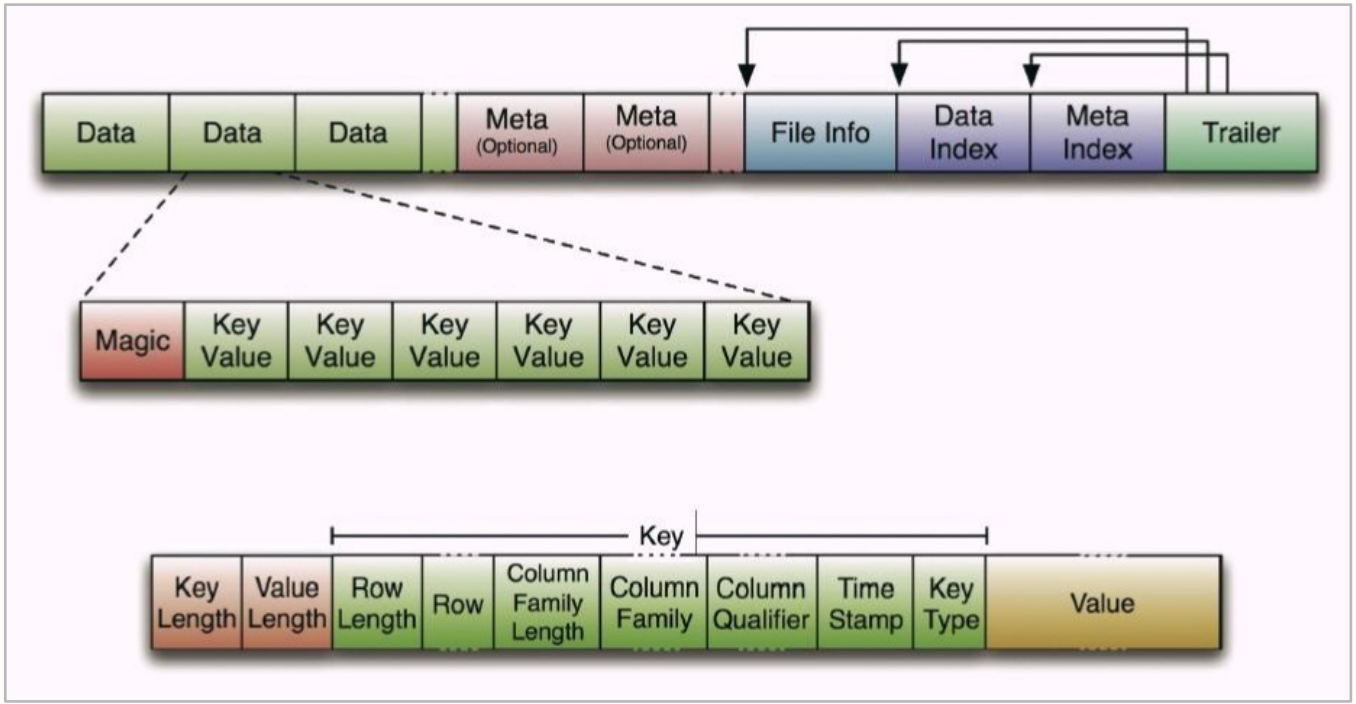
System Architecture: Write-Ahead-Log Flow



System Architecture: WAL (cont.)



System Architecture: HFile and KeyValue



APIs

APIs: Overview

- **Java**
  - Get, Put, Delete, Scan
  - IncrementColumnValue
  - TableInputFormat - MapReduce Source
  - TableOutputFormat - MapReduce Sink
- Rest
- Thrift
- Scala
- Jython
- Python
- Groovy DSL
- Ruby shell
- Java MR, Cascading, Pig, Hive

ACID Properties

ACID Properties

- HBase **not ACID-compliant**, but does guarantee certain specific properties

- **Atomicity**
  - All mutations are atomic within a row. Any put will either wholely succeed or wholely fail.
  - APIs that mutate several rows will *not* be atomic across the multiple rows.
  - The order of mutations is seen to happen in a well-defined order for each row, with no interleaving.

- **Consistency** and **Isolation**
  - All rows returned via any access API will consist of a complete row that existed at some point in the table's history.

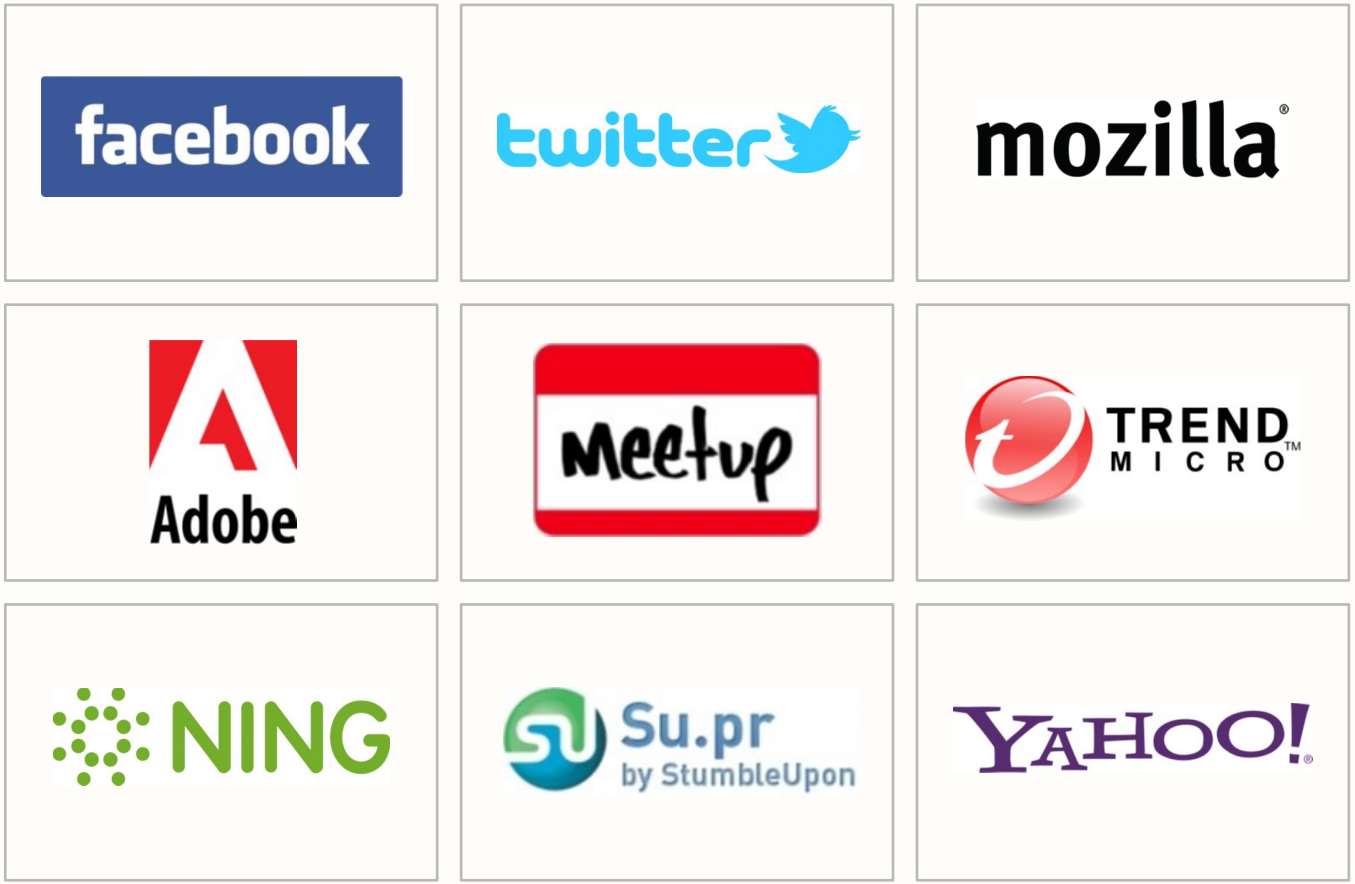
ACID Properties (cont.)

- **Consistency** of Scans
  - A scan is not a consistent view of a table. Scans do not exhibit snapshot isolation.
  - Those familiar with relational databases will recognize this isolation level as "read committed".

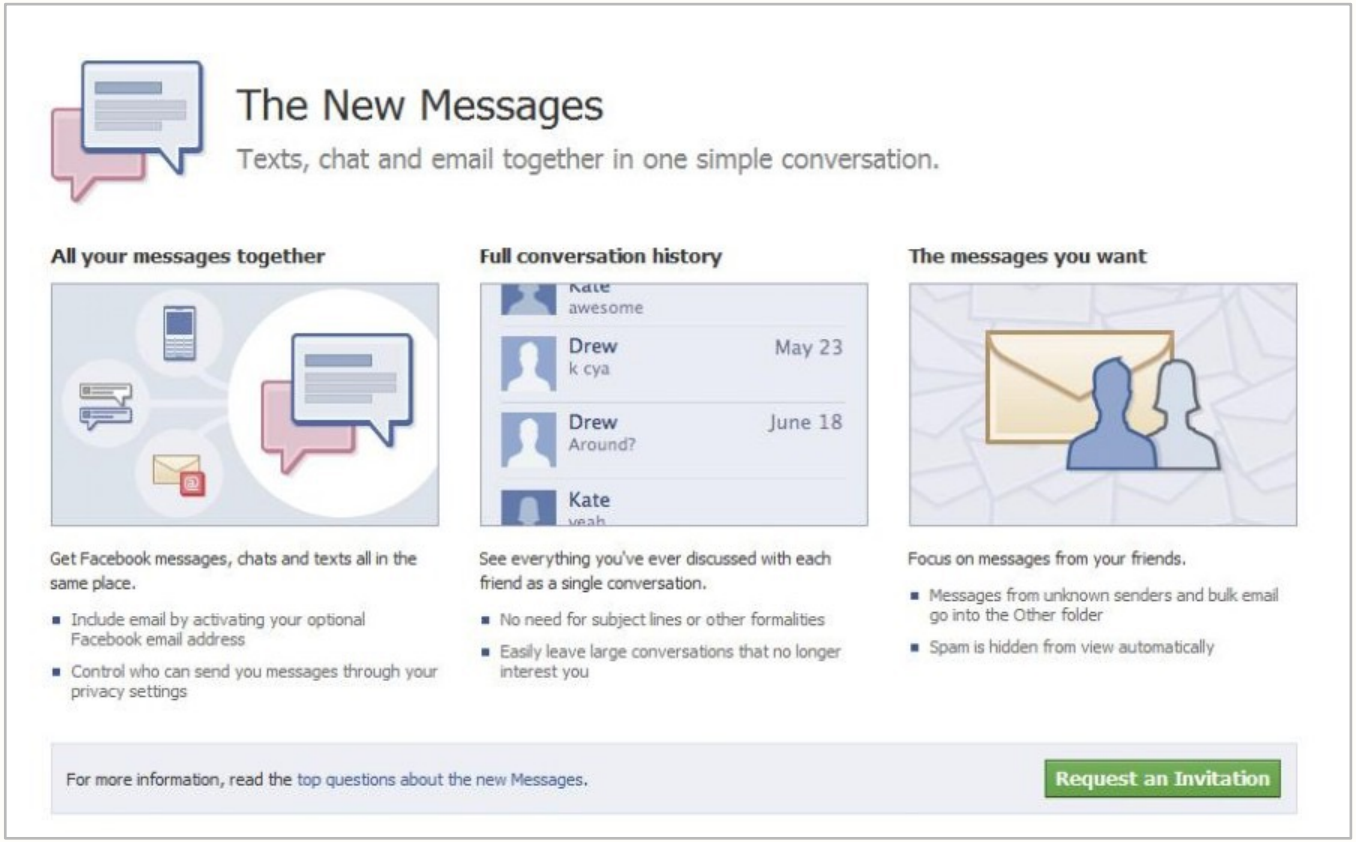
- **Durability**
  - All visible data is also durable data. That is to say, a read will never return data that has not been made durable on disk.
  - Any operation that returns a "success" code (e.g. does not throw an exception) will be made durable.
  - Any operation that returns a "failure" code will not be made durable (subject to the Atomicity guarantees above).
  - All reasonable failure scenarios will not affect any of the listed ACID guarantees.

Users

Users: Just to name a few…

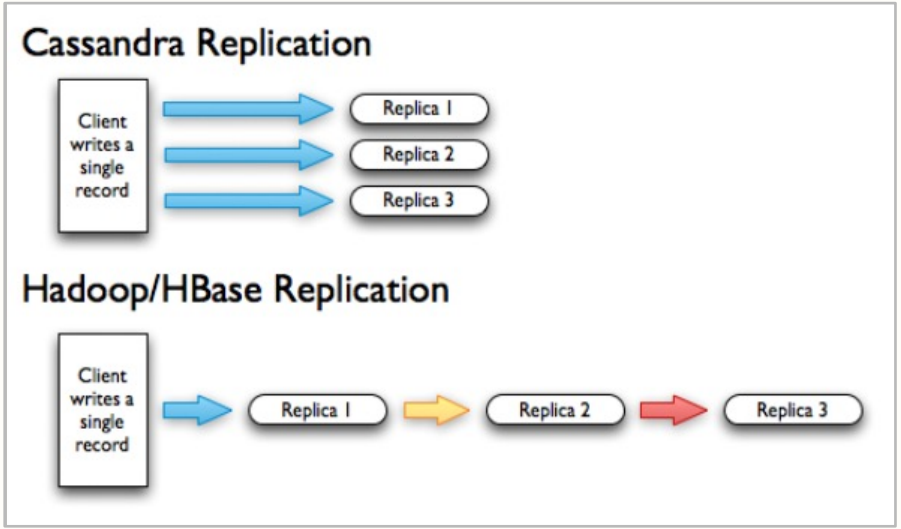


Users: Facebook - Messaging System

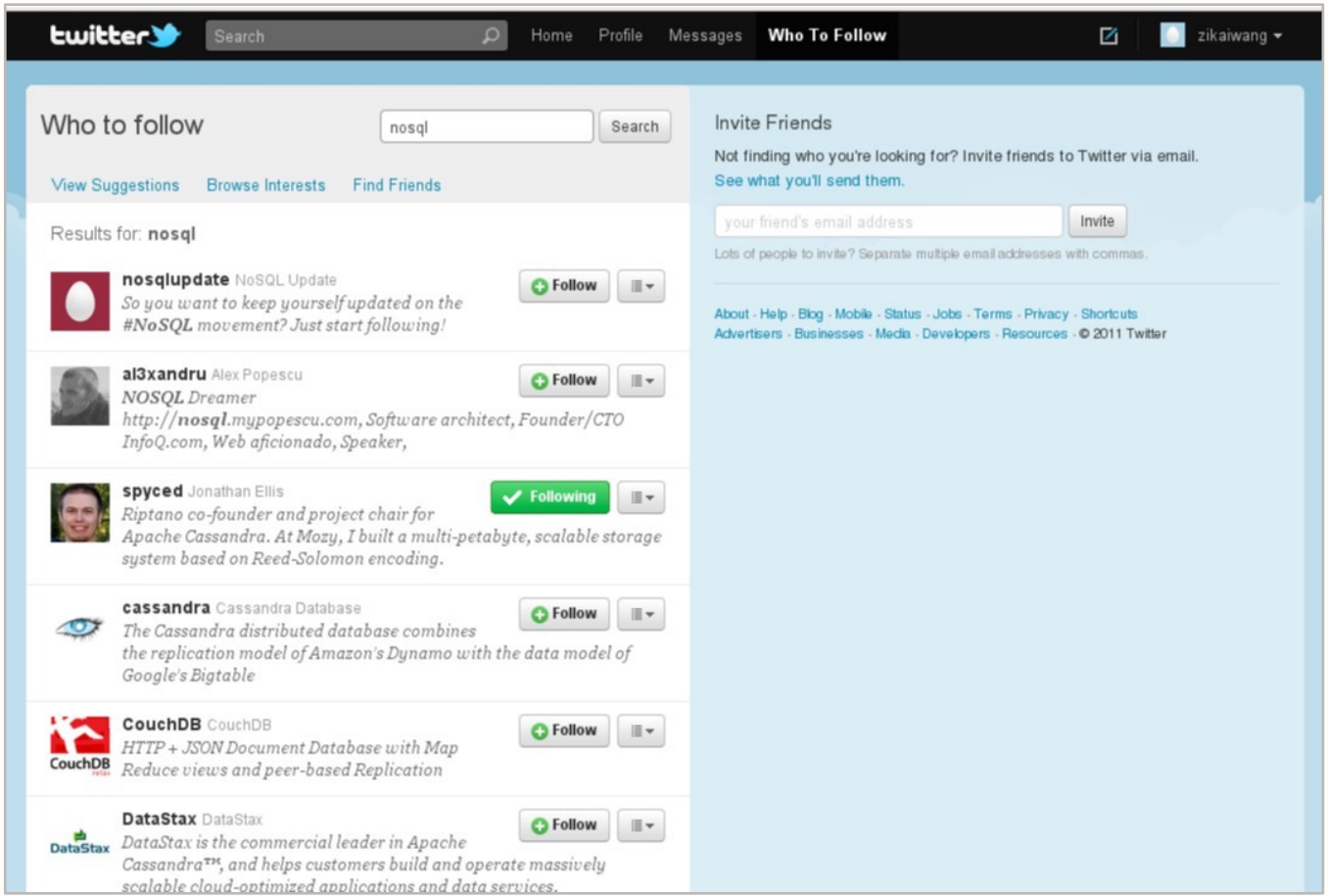


Users: Facebook - Messaging System (cont.)

- **Previous Solution:** Cassandra
- **Current Solution:** HBase
- **Why?** Cassandra's replication behavior



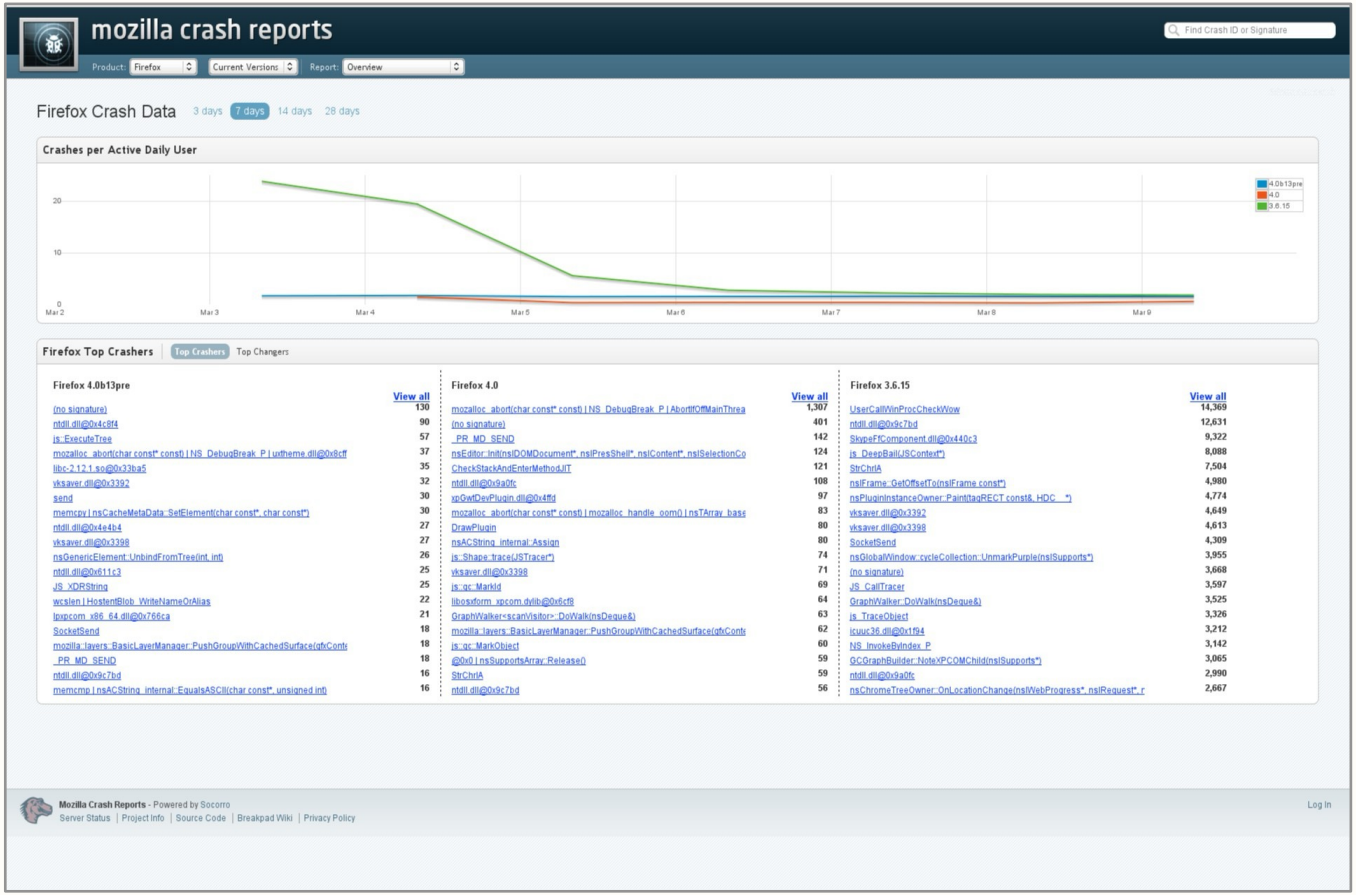
Users: Twitter - People Search



Users: Twitter - People Search (cont.)

- Customer Indexing
- **Previous Solution:** offline process at a single node
- **Current Solution:**
  - Import user data into HBase
  - Periodically MapReduce job reading from HBase
  - Hits FlockDB and other internal services in mapper
  - Write data to sharded, replicated, horizontally scalable, in-memory, low-latency Scala service
- **Vs. Others:**
  - HDFS: Data is mutable
  - Cassandra: OLTP vs. OLAP?

Users: Mozilla - Socorro



Users: Mozilla – Socorro (cont.)

- **Socorro**, Mozilla's crash reporting system (https://crash-stats.mozilla.com/products)
  - Catches, processes, and presents crash data for Firefox, Thunderbird, Fennec, Camino, and Seamonkey.
- 2.5 million crash reports per week, 320GB per day
- **Previous Solution:** NFS (raw data), PostgreSQL (analyze results)
  - 15% of crash reports are processed
- **Current Solution:** Hadoop (processing) + HBase (storage)

HBase vs. RDBMS

HBase vs. RDBMS

HBase	RDBMS
Column-oriented	Row oriented (mostly)
Flexible schema, add columns on the fly	Fixed schema
Good with sparse tables	Not optimized for sparse tables
No query language	SQL
Wide tables	Narrow tables
Joins using MR – not optimized	Optimized for joins (small, fast ones too!)
Tight integration with MR	Not really...

HBase vs. RDBMS (cont.)

HBase	RDBMS
De-normalize your data	Normalize as you can
Horizontal scalability – just add hardware	Hard to shard and scale
Consistent	Consistent
No transactions	Transactional
Good for semi-structured data as well as structured data	Good for structured data