# HBase: Bigtable Goes Realtime

COSCUP 2009, Taipei, Taiwan

# Overview

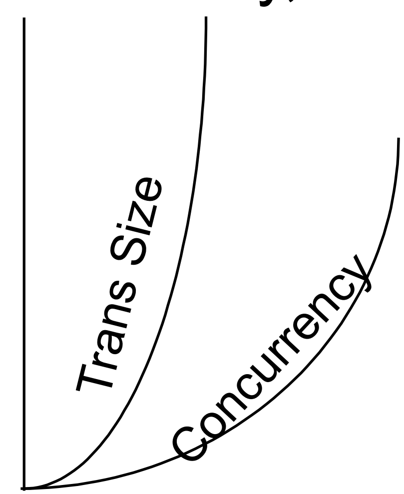
- Bigtable is a response to challenges with the storage and computational aspects of very large data sets
  - Infrastructure scalability issues
  - Disk seek times versus sort-and-merge
  - RDBMS limitations and expense
- The latest HBase release is a faithful Bigtable clone which also focuses on real time responsiveness (random access queries, updates)
- An option for very large scale structured storage
- Features meet the requirements of many "Web 2.0" use cases

# Seek Versus Sort and Merge

- At scale, computation is dominated by disk transfer
  - CPU, RAM, and disk size double every 18-24 months
  - Seek time remains nearly constant (~5% per year)
- Two database paradigms
  - Seek
    - B-Tree (RDBMS): Operates at seek rate – log(N) seeks/access
  - Transfer
    - Sort/merge flat files (MapReduce, Bigtable): Operates at transfer rate – log(N) transfers/sort
- Seek is inefficient compared to transfer at scale
  - Given:
    - 10 MB/second transfer bandwidth
    - 10 milliseconds disk seek time
    - 100 bytes per entry (10 billion entries)
    - 10 kB per page (1 billion pages)
  - Updating 1% of entries (100,000,000) takes:
    - 1,000 days with random B-Tree updates (!!)
    - 100 days with batched B-Tree updates (!)
    - **1 day with sort and merge**

# RDBMS At Scale

- Performance of RDBMS systems is good for transaction processing but not for very large scale analytic processing
- Very large scale analytic processing defined:
  - Big queries – typically range or table scans
  - Big databases (100s or 1,000s of TB)
  - Typical queries exceed the ability of a single server – no matter how large – to process them in a timely manner
- In RDBMS systems, waits and deadlocks rise non-linearly with transaction size and concurrency: Square of concurrency, 3rd power of transaction size




- Sharding is not a solution
  - Application specific
  - Labor intensive (re)partitioning
- Expensive RDBMS systems can deliver large storage capacity and can execute queries over that data which complete in reasonable time, but at a high cost
- Using open source RDBMS scale often requires giving up all relational features (secondary indexes, etc.) for performance

# What If... ?

- Trade relational features for performance, since using a RDBMS at scale often requires giving them up anyway
- Generalize the data model
- Avoid waits and deadlocks by minimizing necessary locking
- Provide transparent horizontal scalability without architectural limits (generic "self sharding")
- Provide fault tolerance and data availability by way of the same mechanisms which allow scalability

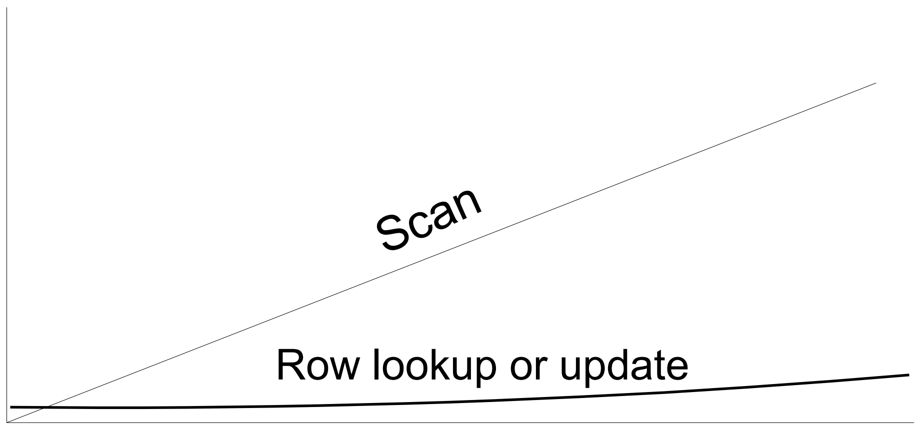
# What is Bigtable?

- http://en.wikipedia.org/wiki/BigTable
- Google: "*Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers [...] a sparse persistent multidimensional sorted map [...]*"
- A column-oriented semi-structured data store
- Strongly consistent
- An engineering response to the storage of massive data sets, data-analytic use cases, and the expense of other solutions

# Bigtable At Scale

- Billions of rows * millions of columns * thousands of versions
  = terabytes or petabytes of storage

- Bigtable systems eschew transactions and relational data models for logarithmic or linear time operations on data despite extreme scale
  - Row key lookup or row mutations are logarithmic
  - Table scans run in linear time
  - No waits or deadlocks



# What is HBase?

- http://hbase.org/

- Created originally at Powerset in 2007

- A faithful clone of Google's proprietary Bigtable architecture, enhanced with additional features developed by the community
  - Fast fault recovery via Zookeeper
  - Query push down via server side query and scanner filters
  - Optimizations for real time queries
    - LRU cache
    - Efficient internal data transfer architecture
  - Rolling restarts
  - Push metrics to log files or Ganglia (http://ganglia.info/)
  - Cell time to lives (TTLs)
  - Administrative GUI and command line shell

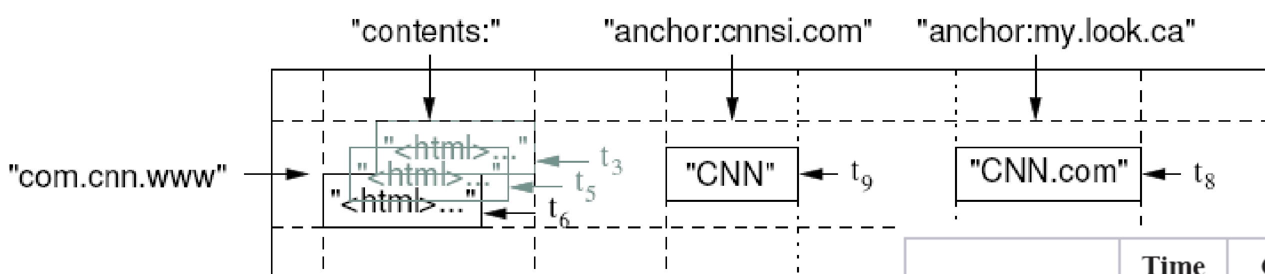
- A Hadoop subproject
  - The usual ASF things apply (license, JIRA, etc)

# What is Bigtable (and HBase) Not?

- Not designed for normalized schemas
  - Tables can have only one index, the row key
  - Secondary indexes can be emulated
- Not a relational query engine, something different
  - Fast (logarithmic time) lookup using row key, optional column and column qualifiers for result set filtering and optional timestamp
  - Full table scan
  - Range scan, with optional timestamp
  - Most recent version or N versions
  - Partial key lookups
    - When combined with compound keys, has the same properties as leading left edge indexes (with the benefit of distributed index)
  - HBase supports server side filters, an extension of the Bigtable architecture, a form of query push down
  - No join operators
    - Run Cascading (http://cascading.org/) on top to recover some relational algebraic operators or do “insert time joins” -- denormalization, view materialization, etc.
- Limited atomicity and transaction support
- Data is unstructured and untyped
- Not accessed or manipulated via SQL

# Data Model

- Distributed multi-dimensional sparse map
- Multidimensional keys: (row, column:qualifier, timestamp)
- Data grouped by columns
- Keys are arbitrary strings
- Access to row data is atomic
- Multiversioning and timestamps avoid edit conflicts caused by concurrent decoupled processes

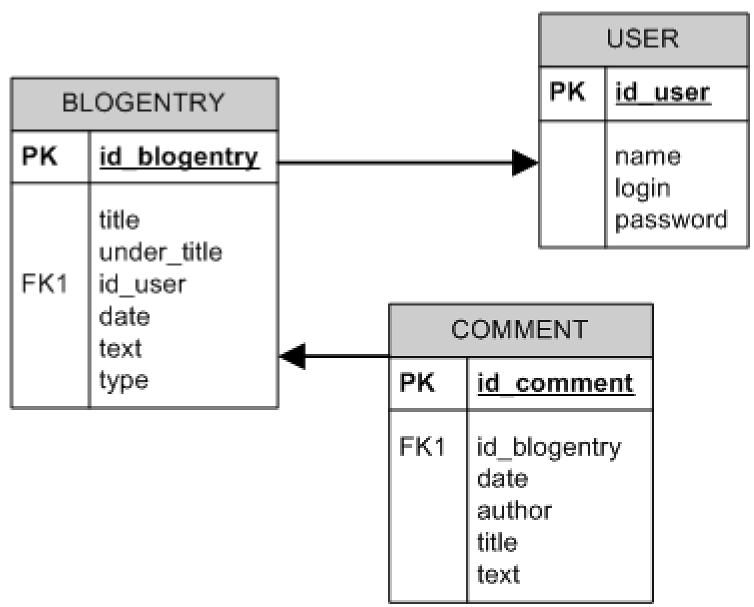



| Row Key | Time Stamp | Column *"contents:"* | Column *"anchor:"* | | Column *"mime:"* |
|---|---|---|---|---|---|
| "com.cnn.www" | t9 | | "anchor:cnnsi.com" | "CNN" | |
| | t8 | | "anchor:my.look.ca" | "CNN.com" | |
| | t6 | "<html>..." | | | "text/html" |
| | t5 | "<html>..." | | | |
| | t3 | "<html>..." | | | |

# Data Model – Real Life Example

- Consider
  - Blog entries, which consist of a title, an under title, a date, an author, a type (or tag), a text, and comments, can be created and updated by logged in users
  - Users, which consist of a username, a password, and a name, can log in and log out
  - Comments, which consist of a title, an author, and text

- Source ERD



# Data Model – Real Life Example (cont.)

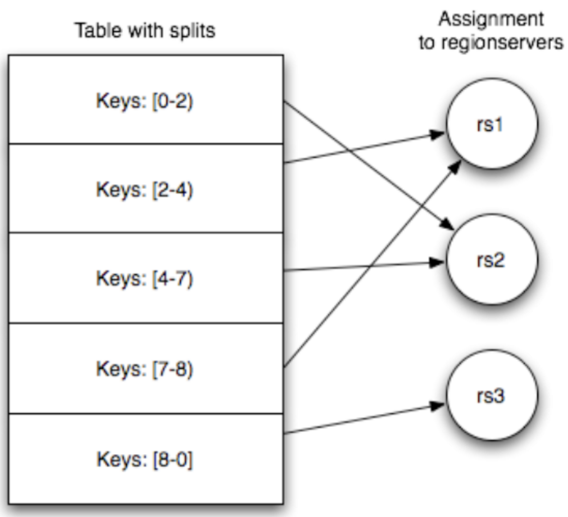
- Target HBase schema

| Table | Row Key | Family | Attributs |
| --- | --- | --- | --- |
| blogtable | TTYYYYMMDDHHmmss | info: | Always contains the column keys author,title,under_title. Should be IN-MEMORY and have a 1 version |
| | | text: | No column key. 3 versions |
| | | comment_title: | Column keys are written like YYYMMDDHHmmss. Should be IN-MEMORY and have a 1 version |
| | | comment_author: | Same keys. 1 version |
| | | comment_text: | Same keys. 1 version |
| usertable | login_name | info: | Always contains the column keys password and name. 1 version |

- Row key
  - A concatenation of type (shortened to 2 letters) and timestamp
  - Rows will be gathered first by type and then by date throughout the cluster
  - More chances of hitting a single region to fetch the needed data
- The one-to-many relationship between BLOGENTRY and COMMENT is handled by storing comments as a family in BLOGENTRY and by using the timestamp as column key
- Building the front page of the blog requires only a fetch of the "info:" family from BLOGTABLE
- By using timestamps in the row key, scanners fetch sequential rows for in order comment rendering

# Tables And Regions

- Rows are stored in byte-lexographic sorted order
- Tables are dynamically split into regions
- Regions are hosted on a number of regionservers
- As regions grow, they are split and distributed evenly among the storage cluster to level load
  - Splits are “almost” instantaneous
  - Fast recovery and fine grained load balancing
  - Regions are migrated away from highly loaded nodes
  - Master rapidly redeploys regions from failed nodes to others

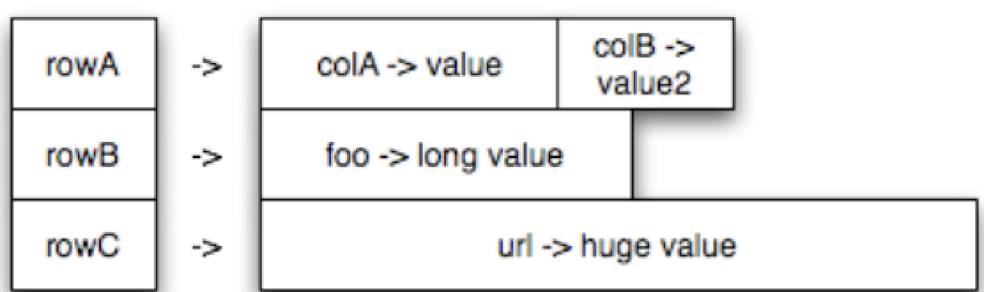



# Column Oriented Storage

- Not a spreadsheet

| | colA | colB | colC | colD |
|---|---|---|---|---|
| rowA | | | | |
| rowB | | | | |
| rowC | | | NULL? | |
| rowD | | | | |

- Instead think of tags



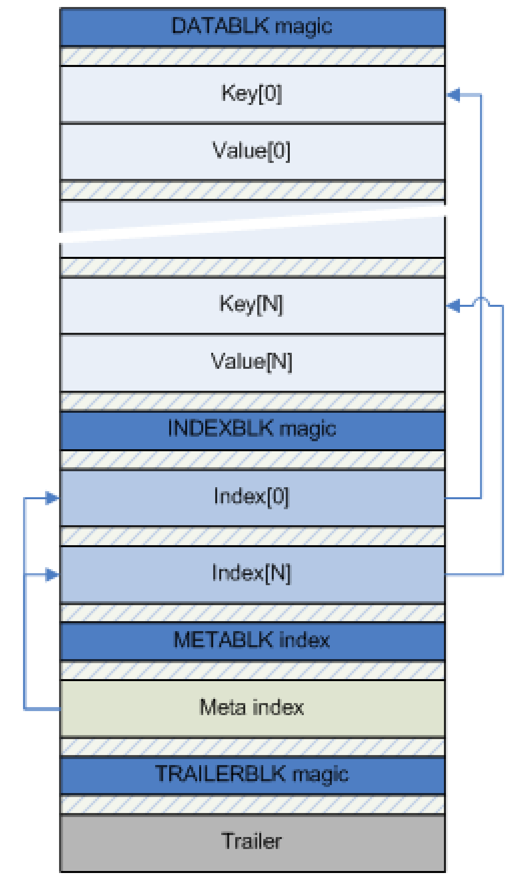

Values of any length, no predefined names or widths

# Column Oriented Storage (cont.)

- A table consists of one or more "column families"
- Column names scoped with an optional qualifier

    *family:qualifier*

  - Effectively, an additional level of indexing
    - First, index into column store to find matching row
    - Then, find value(s) with matching qualifier
- Stored in separate set of files
  - One or more store files for each column family
  - Store files are periodically compacted
  - Values in a column family are stored in sorted order
  - Optional file level compression (GZIP, LZO)
- Lexiographically similar values are packed adjacent to each other into blocks in the column stores and are retrieved most efficiently together
- In general: It is fast and cheap to scan adjacent rows and columns



# Data Consistency

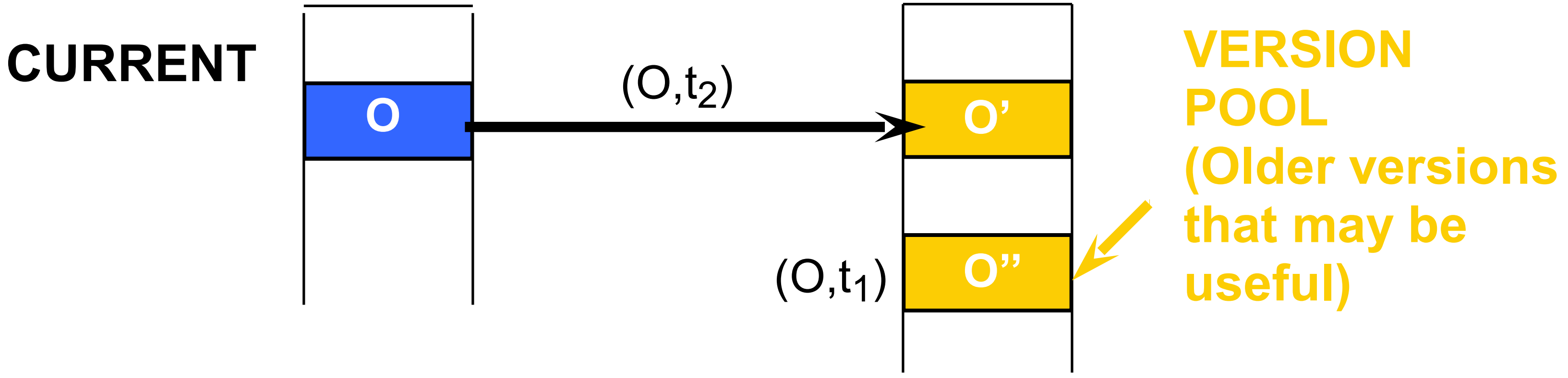
- Data consistency models, strongest to weakest:
  - Strict: Changes are atomic and appear to take effect instantaneously
  - Sequential: Every observer sees all changes in the same order
  - Causal: Changes that are causally related are observed in the same order by all observers
  - Eventual: When no updates occur for a period of time, eventually all updates will propagate through the system and all the replicas will be consistent
  - Weak: No guarantee that all updates will propagate and changes may appear out of order to various observers

- HBase is strictly consistent
  - Every value appears in one region only, within the appropriate boundary [startKey,endKey] for its row key
  - Each region is assigned to only one region server at a time

- HBase's multiversioning and timestamping can help with application layer consistency issues also

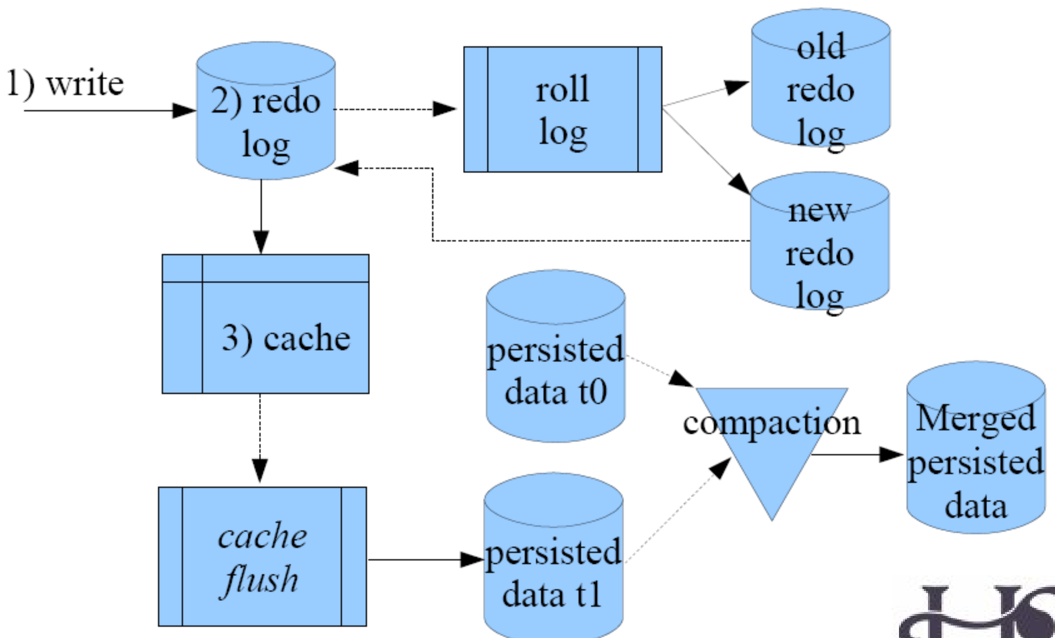
# Data Consistency (cont.)

- HBase's multiversioning and timestamping can help with application layer consistency issues also
  - All edits are timestamped and the storage system supports storage and retrieval of multiple versions of a value
  - No edit will conflict with another, unless applications explicitly set timestamps (in which case conflicts are resolved in favor of the most recent store)
  - Applications can query for the latest version according to timestamp or N versions, depending on requirements



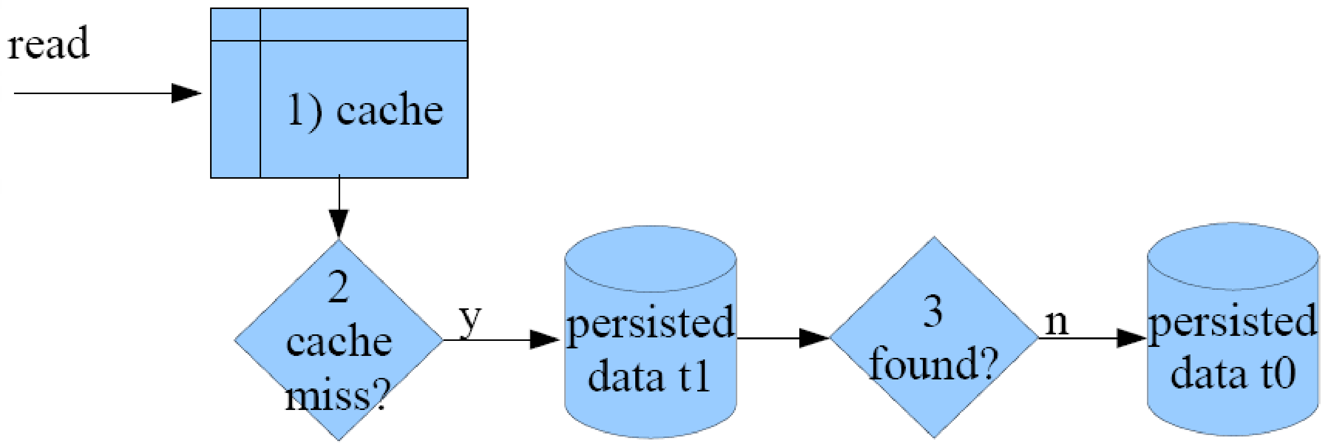
# Write Path

- When a write occurs
  - It is written to a write ahead log
  - It is buffered in memory ("memstore")
  - Deletes are just another kind of write (delete marker)
  - Periodically, the cache is written to disk, creating a new file in each store for the columns being flushed
  - A compaction occurs when the number of files in an store exceeds a threshold: all stores are merge sorted into a single new store file
  - Deleted and expired values are garbage collected during compactions
  - Compactions are done in the background
  - Periodically, the log file is closed and a new one created
  - Old log files are garbage collected
  - As we discussed earlier, updating via rewriting is 100x – 1000x faster than update via seek and replace at large scale



# Read Path

- When a read occurs
  - Check for data in memstore
  - Look for data in persisted data, from newest to oldest
  - Stop backwards search for a given { table, row, column, timestamp (optionally) } coordinate if a delete marker is found
  - Stop search if enough versions have been found to satisfy the search criteria
  - Ignore expired values if TTLs are configured on the relevant column families
  - Expired values will be garbage collected at next compaction, just like deletes



# HBase 0.20 - Performance++

- The first ever performance release
  - Focus on improving performance for
    - Random access time
    - Scan time
    - Insert time
- As a random-access store, we are well suited for the storing and serving of data for Web applications
- However, high latency and variability (100s of ms to seconds) has reduced the usefulness of HBase and required the use of external caching in the past
- Significant performance gains now in 0.20
  - Random read times similar to that of an RDBMS
    - 20 -100 times faster with far less variability
  - Scan times reduced
    - 30 times faster than previous versions
      - 20-30 ms per 500 rows now
      - 300-600 ms per 500 rows before
  - Insert times reduced
    - 2-10 times faster with less than half the memory usage

# HBase 0.20 – Performance++ (cont.)

- Some numbers:
  - Tall table: 1,000,000 rows, 1 column per row (~16 bytes)
    - Sequential insert: 0.024 ms/row
    - Random read: 1.42 ms/row (average)
    - Full scan: 117 ms / 10K rows
  - Wide table: 1,000 rows with 20,000 columns each
    - Sequential insert: 312 ms/row
    - Random reads: 121 ms/row (average)
    - Full scan: 14.6 seconds/100 rows, 146 ms/row
  - Fat Table: 1,000 rows with 10 columns, 1 MB values
    - Sequential insert: 68 ms/row
    - Random reads: 56.92 ms/row (average)
    - Full scan: 3.53 seconds/100 rows, 35ms/row
  - Performance under cache is very good
    - 1 ms to get a single row
    - 20 ms to read 500 rows
    - 75 ms to get 5,500 rows
  - High throughput
    - Import speeds of 10,000 ops/sec/node possible
    - 5,000 ops/sec/node typical for sustained query volume

HBASE

# HBase 0.20 Architectural Improvements

- The Guiding Philosophy – Unjavafy Everything!
  - Zero-copy reads
  - Block-based storage, reading, and indexing
  - Drastically reduce Object instantiation
  - Eliminate widespread usage of Trees
  - Sorted merges using Heap structures
  - Fast and intelligent caching with memory-awareness
- New Key Format – KeyValue
  - Contains only (byte[] buf, int offset, int length)
  - Compact binary format with binary comparators
  - Our “pointer” to keys inside blocks
- New File Format – HFile
  - Originally based on TFile (HADOOP-3315) and BigTable
  - Block based binary format with a block index
  - Persisted storage of List<KeyValue>
- New Block Cache – Concurrent LRU
  - LRU eviction with scan-resistance and block priorities
  - Memory-bound using HeapSize interface
  - Non-blocking and unsynchronized LRU map

H·BASE

# HBase 0.20 Architectural Improvements (cont.)

- New Query API
  - Put, Get, Scan, Delete operations
  - Extended support for versioning
  - Drastically reduced API size and complexity
  - An API that more closely mirrors implementation
- New Result API and optimized serialization
  - Result is just a wrapper for KeyValue[]
  - User-friendly trees are built on-demand, client-side
  - Deserialization allocates a single byte buffer for all KVs in query result
  - Zero-copy sending, single allocation receiving
- New Scanners –KeyValueScanner/ KeyValueHeap
  - Replace linear sort logic with an encapsulated Heap
  - Abstract the handling of versions, deletes, query params now capable of processing individual rows with millions of columns and versions
  - Linear (or worse) to Logarithmic, Logarithmic to Constant

- We improved our performance by more than an order of magnitude in most cases while drastically improving our memory usage

# HBase 0.20 Architectural Improvements (cont.)

- Zookeeper integration
  - http://hadoop.apache.org/zookeeper/
  - A highly available configuration storage system set up in a 2N+1 quorum

  - We now track cluster membership and detect dead servers via ZK
  - HBase supports master election and recovery in multi-master deployments
    - Can kill master and cluster continues operation
    - New or fail over master determines current state and continues
  - Automatic Master failover
  - Rolling upgrades of point releases
  - Modify some cluster configuration without full cluster restart

- No more SPOF in HBase architecture

- Improved REST Web service connectors
  - Integrated REST interface
  - "Stargate" contrib

# HBase 0.20 Extensions

- Transactional tables
  - Transactions can span multiple regions
  - Writes are applied when committing a transaction
  - At commit time, the transaction is examined to see if it can be applied while still maintaining atomicity via Optimistic Concurrency Control
  - Warning: Recovery if region server fails is not fully implemented

- Secondary indexes
  - Maintains a separate index table for each secondary index
  - One or more columns that contribute to the index key
  - Uses transactional layer to maintain consistency

# Roadmap for 0.21 and beyond

- Distribute Master responsibilities further with ZK
  - Further capability to modify configurations at run time
  - State sharing via ZK nodes
  - More distributed/emergent behaviors
- Language-agnostic binary RPC
- Native C/C++ client library
- Multi-DC replication
- Further optimizations on algorithms and data structures
- HBase FSCK
- Intra-row scanning
- Discretionary access control

# HBase At Trend Micro

- Used for threat research
- Used as part of a production platform for event data collection, archiving, and correlation
- A Trend Micro employee is an HBase committer
- Trend Micro is open source friendly

# For More Information

- HBase Website and Wiki
  - http://hbase.org/
  - http://wiki.apache.org/hadoop/Hbase
  - http://wiki.apache.org/hadoop/HBase/HBasePresentations
- Users and supporting projects
  - http://wiki.apache.org/hadoop/Hbase/PoweredBy
  - http://wiki.apache.org/hadoop/SupportingProjects
- Mailing List
  - http://hadoop.apache.org/hbase/mailing_lists.html
- IRC Channel
  - #hbase on Freenode
  - Committers and core contributors are here on a regular basis
  - More active than the Hadoop forums
- Follow us on Twitter!
  - @hbase

H·BASE