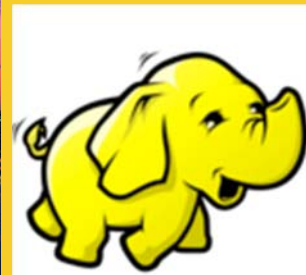




HBase Key Design

Originals of slides and source code for examples: <http://www.coreservlets.com/hadoop-tutorial/>
Also see the customized Hadoop training courses (onsite or at public venues) – <http://courses.coreservlets.com/hadoop-training.html>

Customized Java EE Training: <http://courses.coreservlets.com/>
Hadoop, Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.



For live customized Hadoop training (including prep for the Cloudera certification exam), please email info@coreservlets.com

Taught by recognized Hadoop expert who spoke on Hadoop several times at JavaOne, and who uses Hadoop daily in real-world apps. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
 - JSF 2.2, PrimeFaces, servlets/JSP, Ajax, jQuery, Android development, Java 7 or 8 programming, custom mix of topics
 - Courses available in any state or country. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
 - Spring, Hibernate/JPA, GWT, Hadoop, HTML5, RESTful Web Services

Contact info@coreservlets.com for details



Agenda

- **Storage Model**
- **Querying Granularity**
- **Table Design**
 - Tall-Narrow Tables
 - Flat-Wide Tables

4

Column Family – Storage Unit

- **Column family is how data is separated**
- **Not columns (unlike regular column database)**
- **For a family cells are stored in the same file (HFile) on HDFS**
- **Cells that are not set will not be stored**
 - Nulls are not stored – data must be explicitly set in order to be persisted
 - Coordinate of each cell are be persisted with the value

5

Column Family – Storage Unit

- Rows are stored as a set of cells
- Cells are stored with their coordinates
- Ordered by Row-Id, and then by column qualifier
- Each cell is represented by KeyValue class

```
Row1:ColumnFamily:column1:timestamp1:value1
Row1:ColumnFamily:column2:timestamp1:value2
Row1:ColumnFamily:column3:timestamp2:value3
Row2:ColumnFamily:column1:timestamp1:value4
Row3:ColumnFamily:column1:timestamp1:value5
Row3:ColumnFamily:column3:timestamp1:value6
Row4:ColumnFamily:column2:timestamp3:value7
```

KeyValue(s)

HFile "ColumnFamily/abcdef"

6

HTable – Set of Column Families

```
Row1:ColumnFamily:column1:timestamp1:value1
Row1:ColumnFamily:column2:timestamp1:value2
Row1:ColumnFamily:column3:timestamp2:value3
Row2:ColumnFamily:column1:timestamp1:value4
Row3:ColumnFamily:column1:timestamp1:value5
Row3:ColumnFamily:column3:timestamp1:value6
Row4:ColumnFamily:column2:timestamp3:value7
Row4:ColumnFamily:column3:timestamp4:value8
```

HFile "Family1/abcdefU"

Table will be made of set of Column Families where each is stored in it's own file.

```
Row1:ColumnFamily:column1:timestamp1:value1
Row1:ColumnFamily:column2:timestamp1:value2
Row1:ColumnFamily:column3:timestamp2:value3
Row2:ColumnFamily:column1:timestamp1:value4
Row3:ColumnFamily:column1:timestamp1:value5
Row3:ColumnFamily:column3:timestamp1:value6
Row4:ColumnFamily:column2:timestamp3:value7
Row4:ColumnFamily:column3:timestamp4:value8
```

HFile "Family1/abcdefU"

7

Querying Data From HBase

- **By Row-Id or range of Row Ids**
 - Only load the specific rows with the provided range of ids
 - Only touch the region servers with those ids
 - If no further criteria provided will load all the cells for all of the families which means loading multiple files of HDFS
- **By Column Family**
 - Eliminates the needs to load other storage files
 - Strongly suggested parameter if you know the needed families
- **By Timestamp/Version**
 - Can skip entire store files as the contained timestamp range within a file is considered

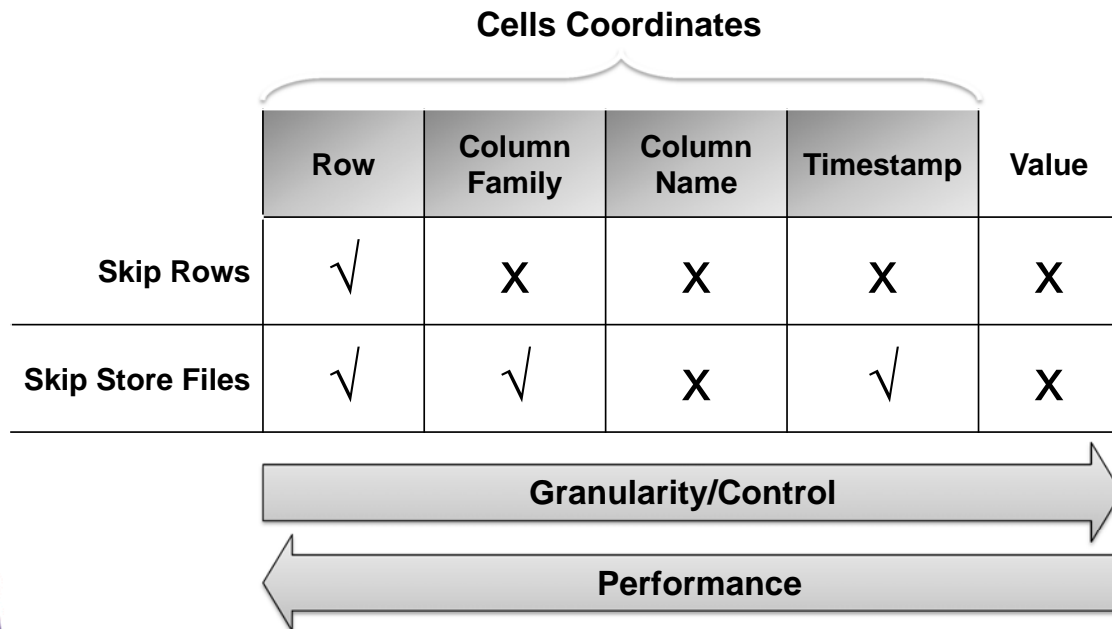
8

Querying Data From HBase

- **By Column Name/Qualifier**
 - Will not eliminate the need to load the storage file
 - Will however eliminate the need to transfer the unwanted data
- **By Value**
 - Can skip cells by using filters
 - The slowest selection criteria
 - Will examine each cell
- **Filters can be applied to each selection criteria**
 - rows, families, columns, timestamps and values
- **Multiple querying criteria can be provided**
 - And most of the time should be provided

9

Querying Granularity vs. Performance



10

Source: Lars, George. [HBase The Definitive Guide](#). O'Reilly Media. 2011

Designing to Store Data in Tables

- **Two options: Tall-Narrow OR Flat-Wide**
- **Tall-Narrow**
 - Small number of columns
 - Large number of rows
- **Flat-Wide**
 - Small number of rows
 - Large number of columns
- **Tall Narrow is generally recommended**
 - Store parts of cell data in the row id
 - Faster data retrieval
 - HBase splits on row boundaries

11

Real World Problem: Blog App

- **Let's say we are implementing a Blog management application**
 - Users will enter blogs
 - Some users will have many blogs and some few
 - You'll want to query by user and date range
- **We will implement simple DataFacade to store and access Blog objects**

12

Real World Problem: Blog App

```
public class Blog {
    private final String username;
    private final String blogEntry;
    private final Date created;
    public Blog(String username, String blogEntry, Date created) {
        this.username = username;
        this.blogEntry = blogEntry;
        this.created = created;
    }
    public String getUsername() {
        return username;
    }
    public String getBlogEntry() {
        return blogEntry;
    }
    public Date getCreated() {
        return created;
    }
    @Override
    public String toString() {
        return this.username + "(" + created + "): " + blogEntry ;
    }
}
```

****Encapsulates username,
Blog entry and the date
the blog was created**

13

Real World Problem: Blog App

- **Two ways to arrange your data**
 - Flat-Wide: each row represents a single user
 - Tall-Narrow: each row represents a single blog, multiple rows will represent a single user
- **Lets assume a simple interface:**

```
public interface DataFacade {  
    public List<Blog> getBlogs(String userId,  
        Date startDate, Date endDate) throws IOException;  
    public void save(Blog blog) throws IOException;  
    public void close() throws IOException;  
}
```

14

1: Flat-Wide Solution

- **All blogs are stored in a single row and family**
- **Each blog is stored in its own column (timestamp1 ... timestamp)**

Row id	entry:t1	entry:t2	...	entry:tN
userid1	Blah blah blah...	Important news..	...	Interesting stuff...
userid2	Interesting stuff...	Blah blah blah...	...	Important news..
userid3	Important news...	Interesting stuff...	...	Blah blah blah...

- **Columns are ordered**
- **Can use filter to select sub-set of columns or page through**

15

1: Flat-Wide Solution

FlatAndWideTableDataFacade.java

```
public class FlatAndWideTableDataFacade implements DataFacade {  
  
    private final HTable table;  
    private final static byte [] ENTRY_FAMILY = toBytes("entry");  
  
    public FlatAndWideTableDataFacade(Configuration conf)  
        throws IOException{  
        table = new HTable(conf, "Blog_FlatAndWide");  
    }  
  
    @Override  
    public void close() throws IOException{  
        table.close();  
    }  
  
    ...  
    ...  
}
```

16

1: Flat-Wide Solution

FlatAndWideTableDataFacade.java

```
@Override  
public void save(Blog blog) throws IOException {  
    Put put = new Put(toBytes(blog.getUsername()));  
    put.add(ENTRY_FAMILY, toBytes(dateToColumn(blog.getCreated())),  
        toBytes(blog.getBlogEntry()));  
    table.put(put);  
}  
  
private String dateToColumn(Date date ){  
    String reversedDateAsStr= Long.toString(Long.MAX_VALUE-date.getTime());  
    StringBuilder builder = new StringBuilder();  
    for ( int i = reversedDateAsStr.length(); i < 19; i++){  
        builder.append('0');  
    }  
    builder.append(reversedDateAsStr);  
    return builder.toString();  
}  
  
public Date columnToDate(String column){  
    long reverseStamp = Long.parseLong(column);  
    return new Date(Long.MAX_VALUE-reverseStamp);  
}
```

entry:timestamp format, each blog is stored in it's own column; for a user all blogs are stored in the same row

Encode and decode date into a column name; columns are ordered, reversing timestamp and padding will force latest records to be retrieved first

17

1: Flat-Wide Solution

FlatAndWideTableDataFacade.java

```
@Override
public List<Blog> getBlogs(String userId, Date startDate, Date endDate) throws IOException {
    Get get = new Get(toBytes(userId)); ← Get row by user id

    FilterList filters = new FilterList();
    filters.addFilter(new QualifierFilter(CompareOp.LESS_OR_EQUAL,
        new BinaryComparator(toBytes(dateToColumn(startDate))));
    filters.addFilter(new QualifierFilter(CompareOp.GREATER_OR_EQUAL,
        new BinaryComparator(toBytes(dateToColumn(endDate)))); ← Utilize filter mechanism to only retrieve provided
    get.setFilter(filters);                                     date range; this works because timestamp is
                                                            encoded in the column name

    Result result = table.get(get);
    Map<byte[], byte[]> columnValueMap = result.getColumnValueMap(ENTRY_FAMILY);

    List<Blog> blogs = new ArrayList<Blog>();
    for (Map.Entry<byte[], byte[]> entry : columnValueMap.entrySet()){
        Date createdOn = toDate(Bytes.toString(entry.getKey()));
        String blogEntry = Bytes.toString(entry.getValue());
        blogs.add(new Blog(userId, blogEntry, createdOn));
    }
    return blogs;
}
```

18

2: Tall-Narrow Solution

- 1 blog per row
- Row id will be comprised of user and timestamp

Row id	entry:blog
userid1_t1	Blah blah blah...
userid1_t2	Interesting stuff...
...	...
userid1_tN	Important news...
....	...
....	...
userid3_500	Blah blah blah...
userid3_501	Blah blah blah...

19

2: Tall-Narrow Solution TallAndNarrowTableDataFacade.java

```
public class TallAndNarrowTableDataFacade implements DataFacade {  
  
    private final HTable table;  
    private final static byte [] ENTRY_FAMILY = toBytes("entry");  
    private final static byte [] BLOG_COLUMN = toBytes("blog");  
    private final static byte [] CREATED_COLUMN = toBytes("created");  
    private final static byte [] USER_COLUMN = toBytes("user");  
  
    private final static char KEY_SPLIT_CHAR = '_';  
    public TallAndNarrowTableDataFacade(Configuration conf)  
        throws IOException{  
        table = new HTable(conf, "Blog_TallAndNarrow");  
    }  
  
    @Override  
    public void close() throws IOException {  
        table.close();  
    }  
  
    ...  
    ...  
}
```

20

2: Tall-Narrow Solution TallAndNarrowTableDataFacade.java

```
@Override  
public void save(Blog blog) throws IOException {  
  
    Put put = new Put(toBytes(blog.getUsername() + KEY_SPLIT_CHAR +  
        convertForId(blog.getCreated())));  
  
    put.add(ENTRY_FAMILY, USER_COLUMN, toBytes(blog.getUsername()));  
    put.add(ENTRY_FAMILY, BLOG_COLUMN, toBytes(blog.getBlogEntry()));  
    put.add(ENTRY_FAMILY, CREATED_COLUMN,  
        toBytes(blog.getCreated().getTime()));  
  
    table.put(put);  
}
```

Encode username and creation timestamp into key

Store blog in and other metadata in column(s)

Save one blog per row

21

2: Tall-Narrow Solution TallAndNarrowTableDataFacade.java

```
private String convertForId(Date date ){
    return convertForId(date.getTime());
}

private String convertForId(long timestamp){
    String reversedDateAsStr=
        Long.toString(Long.MAX_VALUE-timestamp);
    StringBuilder builder = new StringBuilder();
    for ( int i = reversedDateAsStr.length(); i < 19; i++){
        builder.append('0');
    }
    builder.append(reversedDateAsStr);
    return builder.toString();
}
```

Encode date into a string to be used for row_id; ids are ordered, reversing timestamp and padding will force latest records to be retrieved first

22

2: Tall-Narrow Solution TallAndNarrowTableDataFacade.java

```
@Override
public List<Blog> getBlogs(String userId, Date startDate,
    Date endDate) throws IOException {
    List<Blog> blogs = new ArrayList<Blog>();

    Scan scan = new Scan(toBytes(userId + KEY_SPLIT_CHAR +
        convertForId(endDate)), toBytes(userId + KEY_SPLIT_CHAR +
        convertForId(startDate.getTime()-1)));

    scan.addFamily(ENTRY_FAMILY);

    ResultScanner scanner = table.getScanner(scan);
    for (Result result : scanner){
        String user = Bytes.toString(
            result.getValue(ENTRY_FAMILY, USER_COLUMN));
        String blog = Bytes.toString(
            result.getValue(ENTRY_FAMILY, BLOG_COLUMN));
        long created = toLong(
            result.getValue(ENTRY_FAMILY, CREATED_COLUMN));
        blogs.add(new Blog(user, blog, new Date(created)));
    }
    return blogs;
}
```

Carefully construct start and stop keys of the scan; recall that row id is userId+reversed timestamp

Each row equals a blog

23

TableDesignExample.java

```
public class TableDesignExample {
    private final static DateFormat DATE_FORMAT = new
        SimpleDateFormat("yyyy.MM.dd hh:mm:ss");

    public static void main(String[] args) throws IOException {
        List<Blog> testBlogs = getTestBlogs();
        Configuration conf = HBaseConfiguration.create();

        DataFacade facade = new FlatAndWideTableDataFacade(conf);
        printTestBlogs(testBlogs);
        exerciseFacade(facade, testBlogs);
        facade.close();

        facade = new TallAndNarrowTableDataFacade(conf);
        exerciseFacade(facade, testBlogs);
        facade.close();
    }
}
```

Try both facades by running through the same code

24

TableDesignExample.java

```
private static void printTestBlogs(List<Blog> testBlogs) {
    System.out.println("-----");
    System.out.println("Test set has [" +
        testBlogs.size() + "] Blogs:");
    System.out.println("-----");
    for (Blog blog : testBlogs){
        System.out.println(blog);
    }
}

private static List<Blog> getTestBlogs(){
    List<Blog> blogs = new ArrayList<Blog>();
    blogs.add(new Blog("user1", "Blog1", new Date(11111111)));
    blogs.add(new Blog("user1", "Blog2", new Date(22222222)));
    blogs.add(new Blog("user1", "Blog3", new Date(33333333)));
    blogs.add(new Blog("user2", "Blog4", new Date(44444444)));
    blogs.add(new Blog("user2", "Blog5", new Date(55555555)));
    return blogs;
}
```

25

TableDesignExample.java

```
public static void exerciseFacade(DataFacade facade,
    List<Blog> testBlogs) throws IOException{

    System.out.println("-----");
    System.out.println("Running against facade: " +
        facade.getClass());
    System.out.println("-----");

    for (Blog blog : testBlogs){
        facade.save(blog);
    }

    getBlogs(facade, "user1",
        new Date(22222222), new Date(33333333));
    getBlogs(facade, "user2",
        new Date(55555555), new Date(55555555));
}
```

26

TableDesignExample.java

```
private static void getBlogs(DataFacade facade,
    String user, Date start, Date end)throws IOException {

    System.out.println("Selecting blogs for user [" + user + "] " +
        "between [" + DATE_FORMAT.format(start) + "] " +
        "and [" + DATE_FORMAT.format(end) + "]");

    for ( Blog blog : facade.getBlogs(user, start, end)){
        System.out.println(" " + blog);
    }
}
```

27

Run TableDesignExample

```
$ yarn jar $PLAY_AREA/HadoopSamples.jar \  
    hbase.tableDesign.TableDesignExample
```

28

TableDesignExample's Output

Test set has [5] Blogs:

```
-----  
[Blog1] by [user1] on [1969.12.31 10:05:11]  
[Blog2] by [user1] on [1970.01.01 01:10:22]  
[Blog3] by [user1] on [1970.01.01 04:15:33]  
[Blog4] by [user2] on [1970.01.01 07:20:44]  
[Blog5] by [user2] on [1970.01.01 10:25:55]  
-----
```

Running aggainst facade: class hbase.tableDesign.**FlatAndWideTableDataFacade**

```
-----  
Selecting blogs for user [user1] between [1970.01.01 01:10:22] and [1970.01.01 04:15:33]  
  [Blog3] by [user1] on [1970.01.01 04:15:33]  
  [Blog2] by [user1] on [1970.01.01 01:10:22]  
Selecting blogs for user [user2] between [1970.01.01 10:25:55] and [1970.01.01 10:25:55]  
  [Blog5] by [user2] on [1970.01.01 10:25:55]  
-----
```

Running aggainst facade: class hbase.tableDesign.**TallAndNarrowTableDataFacade**

```
-----  
Selecting blogs for user [user1] between [1970.01.01 01:10:22] and [1970.01.01 04:15:33]  
  [Blog3] by [user1] on [1970.01.01 04:15:33]  
  [Blog2] by [user1] on [1970.01.01 01:10:22]  
Selecting blogs for user [user2] between [1970.01.01 10:25:55] and [1970.01.01 10:25:55]  
  [Blog5] by [user2] on [1970.01.01 10:25:55]
```

29

Flat-Wide vs. Tall-Narrow

- **Tall Narrow has superior query granularity**
 - query by rowid will skip rows, store files!
 - Flat-Wide has to query by column name which won't skip rows or storefiles
- **Tall Narrow scales better**
 - HBase splits at row boundaries meaning it will shard at blog boundary
 - Flat-Wide solution only works if all the blogs for a single user can fit into a single row
- **Flat-Wide provides atomic operations**
 - atomic on per-row-basis
 - There is no built in concept of a transaction for multiple rows or tables

30

© 2012 coreservlets.com and [Dima May](#)



Wrap-Up

Customized Java EE Training: <http://courses.coreservlets.com/>

Hadoop, Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Android.

Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

Summary

- **We learned**
 - How Data Stored
 - Querying Granularity
 - Tall-Narrow Tables
 - Flat-Wide Tables

32

© 2012 coreservlets.com and [Dima May](#)



Questions?

More info:

<http://www.coreservlets.com/hadoop-tutorial/> – Hadoop programming tutorial
<http://courses.coreservlets.com/hadoop-training.html> – Customized Hadoop training courses, at public venues or onsite at *your* organization
<http://courses.coreservlets.com/Course-Materials/java.html> – General Java programming tutorial
<http://www.coreservlets.com/java-8-tutorial/> – Java 8 tutorial
<http://www.coreservlets.com/JSF-Tutorial/jsf2/> – JSF 2.2 tutorial
<http://www.coreservlets.com/JSF-Tutorial/primefaces/> – PrimeFaces tutorial
<http://coreservlets.com/> – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training

Customized Java EE Training: <http://courses.coreservlets.com/>
Hadoop, Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.