COMP5349 – Cloud Computing

Week 6: Data Analytics in the Cloud

A/Prof Dr Uwe Röhm
School of Information Technologies

Outline

- **Motivation**
- Approaches to Data Analytics with Map/Reduce
  - Pig Latin
  - HIVE
  - (Tenzing) (only briefly)
- Conclusions

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Motivation

- Large Internet Companies
  - E.g. Amazon, Google, Yahoo!, Microsoft, Facebook
- collecting terabytes of data each day
- Need ad-hoc analysis of this information
  - together with the historical trends
  - Example:
    - Engineers analyzing the search log for trends to improve the search engine's ranking algorithm

Why SQL is not seen as 'The Solution'

- Costs:
  - Traditional data warehouses are seen as too expensive (TCO-wise)
  - E.g. typical prizing models are per CPU/core which gets too expensive for the massively parallel systems of Internet companies
- Programming Style
  - SQL seen as 'impedance mismatch' by programmers
- Schema-First Approach
  - RDBMS require data to be loaded into own infrastructure for ensuring transactional consistency, indexing and data curating
  - Such data import into a DBMS 'overkill' for temporary data sets
    - Mainly scans over whole data set, basically no point lookups and read-only data

Is Map/Reduce ‘The Solution’?

- Map/Reduce paradigm is very powerful to tackle the peta-byte scale problems of today’s major players
  - E.g. Google, Amazon, Yahoo! or Facebook
- Pros:
  - Scalability and runs on commodity hardware
- Cons:
  - Not usable for non-programmers
  - Even non-procedural programmers struggle with the functional nature of Map/Reduce
- Big Question:
  - Is there something between Map/Reduce and SQL?

Short-Comings of Map/Reduce

- Simple (key, value) format
  - How to handle multi-column data / data with structure?
- Single-input, two-stage data flow
  - How to do joins between different inputs?
  - How to program multi-stage analysis tasks?
- Programming Abstraction
  - map() and reduce() functions assume a functional paradigm which is hard to comprehend for non-programmers
  - Even standard functions such as projections or filtering first have to be programmed as a map() or reduce() function

=> Difficult to maintain and to reuse code

Approaches

- General Idea:
  - Highly-parallel shared-nothing cluster infrastructure
  - Map/Reduce infrastructure (or Hadoop)
  - with analytical layer on top

- Several Approaches
  - Yahoo!: Pig (SIGMOD2008)
  - Facebook: HIVE (ICDE2010)

  - Similar tools from Google:
    - HIVE -> Google Tenzing (VLDB2011)
    - Pig -> Google Sawzall (Scientific Programming, 2005)

Pig (Latin)

Apache Pig [SIGMOD2008]

- 'data-flow' language layer on top of Hadoop
  - combining declarative querying and map/reduce
  - **Pig** is the platform. The language for the platform is called **Pig Latin**
- Sequence of data transformation steps
- With a couple of pre-defined high-level operations
  - Filtering
  - Grouping
  - Aggregation
- Some similarity to manually write a query execution plan…
  - In consequence: no optimizer (at least not as of 2008)
- available as Apache Pig project (pig.apache.org)
  - originally developed at Yahoo! Research labs

看到这里

Pig Example

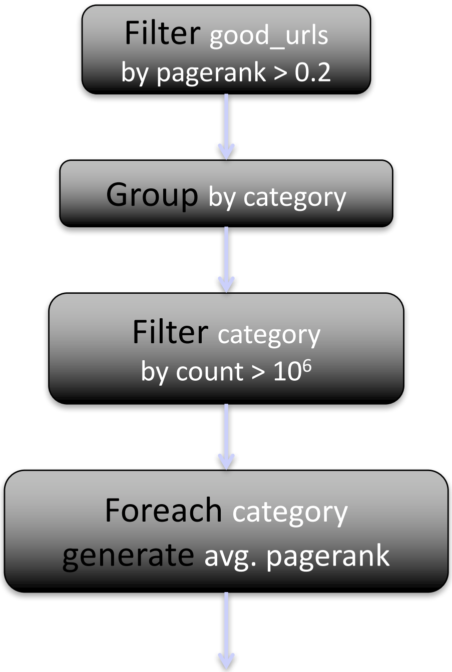
- Given Table: Urls (url, category, pagerank)
- In SQL:

  SELECT category, AVG(pagerank)
  FROM Urls
  WHERE pagerank > 0.2
  GROUP BY category **HAVING** COUNT(*) > 10^6

- In Pig Latin:

  good_urls = **FILTER** urls **BY** pagerank > 0.2;
  groups = **GROUP** good_urls **BY** category;
  big_groups = **FILTER** groups **BY** COUNT(good_urls)>10^6;
  output = **FOREACH** big_groups
  GENERATE category, AVG(good_urls.pagerank);

Pig Latin: Dataflow for Example Query



Pig Latin: Schemas

- In Pig Latin, schemas are optional
- Instead, the i^{th} field of a file can be addressed directly

- Example:
- Instead of
  good_urls = **FILTER** urls **BY** pagerank > 0.2;
  also possible to check for the third field
  good_urls = **FILTER** urls **BY** $2 > 0.2;

Pig Latin's Data Model

- fully nested data model:
  - Rows with multiple columns of pre-defined types
  - Atomic data types
    - Integers, floats, strings
  - Complex Data types
    - Associative arrays [*key-type* -> *value-type*]
    - Sets { *element-type* }
    - Tuples (*field-type*$_1$, *field-type*$_2$, …, *field-type*$_n$)
    - Nested types
  - Missing values: NULL supported too

- Allows programmers to write UDFs based on (complex) types

t = ('alice', { ('lakers', 1) ('iPod', 2) }, ['age' → 20])

Let fields of tuple t be called f1, f2, f3

Expression Type	Example	Value for t
Constant	'bob'	Independent of t
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{ ('lakers') ('iPod') }
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

Expressions in Pig Latin

Pig Latin Commands

- Specifying Input Data: LOAD

queries = **LOAD** ‘query_log.txt’
USING myLoad()
AS (userId, queryString, timestamp);

- Per-tuple Processing: FOREACH

expand_queries = **FOREACH** queries **GENERATE**
userId, expandQuery(queryString);

Pig Latin Commands (Cont.)

- Discarding Unwanted Data: FILTER

real_queries = **FILTER** queries **BY** userId **neq** ‘bot’;
or **FILTER** queries **BY NOT** isBot(userId);

- Filtering conditions involve combination of expression, comparison operators such as ==, eq, !=, neq, and the logical connectors AND, OR, NOT

Pig Latin Commands (Cont.)

- **Getting Related Data Together:** COGROUP
  Suppose we have two data sets
  result: (queryString, url, position)
  revenue: (queryString, adSlot, amount)

  grouped_data = COGROUP result BY queryString,
  revenue BY queryString;

Pig Latin Commands (Cont.)

- GROUP
  grouped_revenue = **GROUP** revenue BY queryString;
  query_revenue = **FOREACH** *grouped_revenue*
  GENERATE queryString,
  SUM(revenue.amount) **AS**
  totalRevenue;

- JOIN in Pig Latin
  join_result = **JOIN** result **BY** queryString ,
  revenue **BY** queryString;

Pig Latin Commands (Cont.)

- Example: Map-Reduce in Pig Latin
  - Assume two user-defined functions map(),reduce()

map_result = **FOREACH** input **GENERATE**
FLATTEN(map(*));

key_group = **GROUP** *map_result* **BY** $0;

output = **FOREACH** *key_group* **GENERATE** reduce(*);

Pig Latin Commands (Cont.)

- **Other Command**
  UNION : Returns the union of two or more bags
  CROSS: Returns the cross product
  ORDER: Orders a bag by the specified field(s)
  DISTINCT: Eliminates duplicate tuple in a bag

- **Nested Operations**
  Pig Latin allows some commands to be nested within a FOREACH command

Example 2: Data Analysis Task

Find the top 10 most visited pages in each category

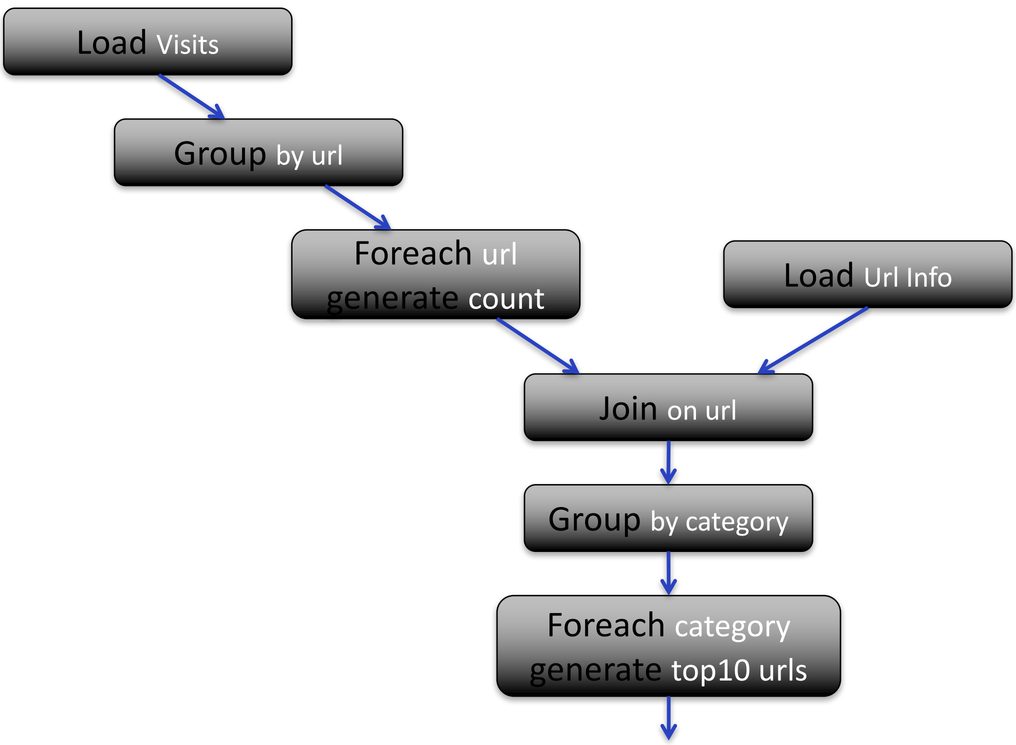
Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

Url Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

Data Flow for Example 2

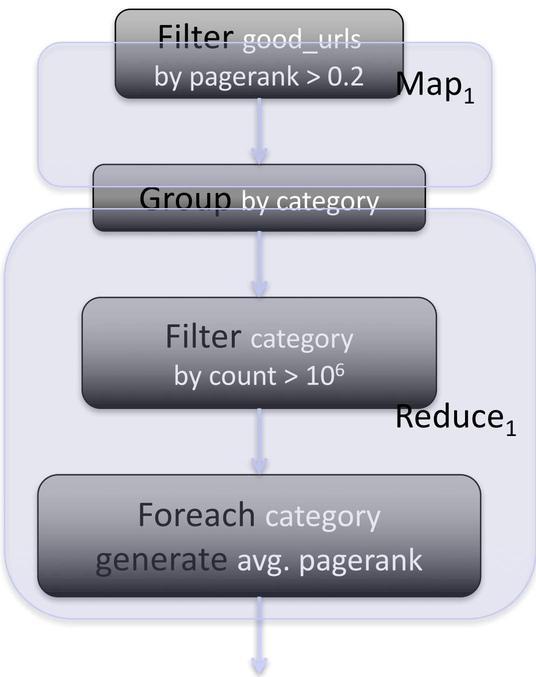


In Pig Latin

```
visits        = load '/data/visits' as (user, url, time);
gVisits       = group visits by url;
visitCounts   = foreach gVisits generate url, count(visits);

urlInfo       = load '/data/urlInfo' as (url, category, pRank);
visitCounts   = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';
```

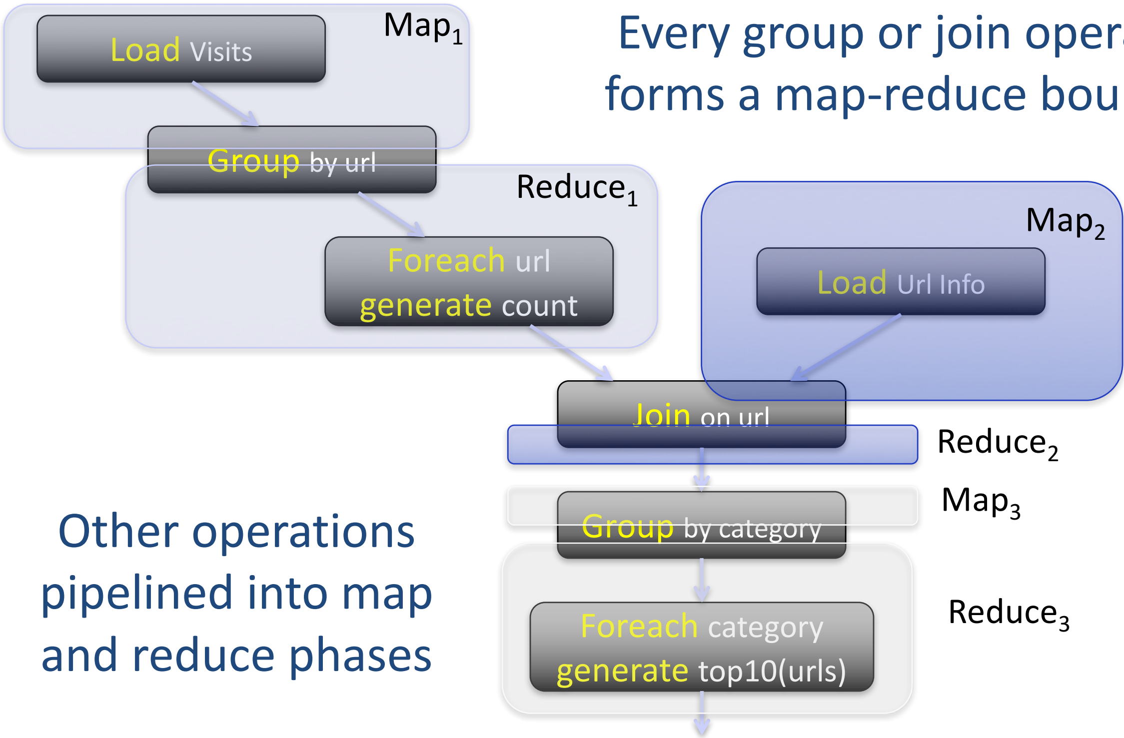

Compilation into Map-Reduce: Example 1

Every group or join operation forms a map-reduce boundary



Other operations pipelined into map and reduce phases

Compilation into Map-Reduce: Example 2



Every group or join operation forms a map-reduce boundary

Other operations pipelined into map and reduce phases

Pig versus SQL

- Pig
  - Uses lazy evaluation
  - Follows an ETL approach: extract, transform, load
  - Is able to store data at any point during the dataflow pipeline
  - Pig declares execution plans, while SQL declares result sets
    - Pig Latin is *procedural*
    - SQL is *declarative*
  - Supports pipeline splits, that workflows can become (acyclic) DAGs while in SQL it sequentially evaluated and produces only a single result

HIVE [ICDE2010]

The Motivation for HIVE

- Scalable analysis on large data sets has been core to the functions of a number of teams at Facebook
- Made Open Source (via Apache Foundation)
  - Also used by CNET, Digg, Chitika, eHarmony, …
    (from https://cwiki.apache.org/confluence/display/Hive/PoweredBy)
- What the data and analysis job look like
  - Structured logs with rich data types (structs, lists and maps)
  - A user base wanting to access this data in the language of their choice
  - A lot of traditional SQL workloads on this data (filters, joins and aggregations)
  - Other non SQL workloads

The Motivation for HIVE (cont'd)

- The possible solutions
  - **Commercial RDBMS** was seen as inadequate to handle faster growing data
    - From 15TB in 2007 to 700TB data in 2009
    - 5 TB of data added each day
    - Some daily data processing job took more than a day to process…
  - **Hadoop MR framework** is scalable and suitable for the type of data analysis
    - Too low level
    - Not efficient for users to write/repeat MR program for simple, standard analysis tasks
    - Many data analysis jobs can be expressed in SQL, which has a large user base
    - Side point: data when stored with 3-way replication in Hadoop: 2.1 PB
- The ideal solution: SQL + MR = HIVE (started 2008)

Usage Scenario of HIVE at Facebook

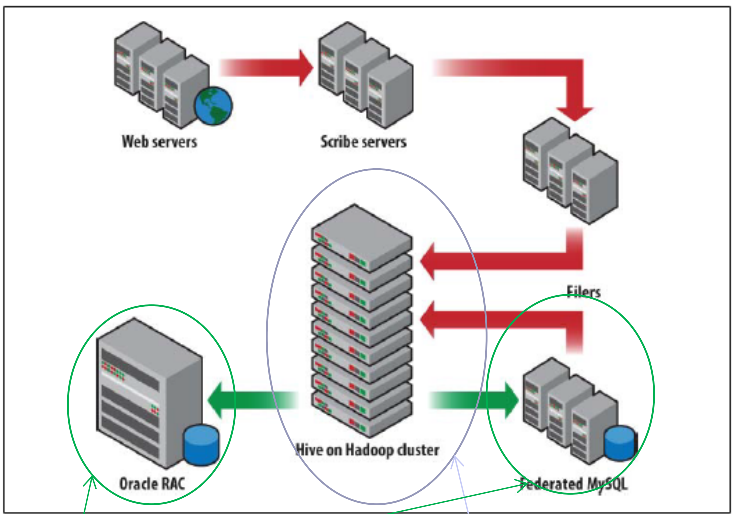


Figure 14-4. Data warehousing architecture at Facebook

Offline batch processing

Online querying of the summary data

Diagram from Tom White, Hadoop, the definitive Guide, O'Reilly, 2009, page 416

Usage Scenario Facebook (ICDE 2010)

facebook

Hadoop & Hive Cluster @ Facebook

- **Production Cluster**
  - 300 nodes/2400 cores
  - 3PB of raw storage
- **Adhoc Cluster**
  - 1200 nodes/9600 cores
  - 12PB of raw storage
- **Node (DataNode + TaskTracker) configuration**
  - 2CPU, 4 core per cpu
  - 12 x 1TB disk (900GB usable per disk)

Slide from Facebook presentation at ICDE 2010 (http://www.slideshare.net/ragho/hive-icde-2010)

Usage Statistics Facebook (ICDE 2010)

facebook

Hive & Hadoop Usage @ Facebook

- **Statistics per day:**
  - 10TB of compressed new data added per day
  - 135TB of compressed data scanned per day
  - 7500+ Hive jobs per day
  - 80K compute hours per day
- **Hive simplifies Hadoop:**
  - New engineers go though a Hive training session
  - ~200 people/month run jobs on Hadoop/Hive
  - Analysts (non-engineers) use Hadoop through Hive
  - 95% of hadoop jobs are Hive Jobs

Slide from Facebook presentation at ICDE 2010 (http://www.slideshare.net/ragho/hive-icde-2010)

HIVE [ICDE2010]

- **A database/data warehouse on top of Hadoop**
  - Structured data similar to relational schema
    - Tables, columns, rows and partitions
    - Support for a variety of basic data types
  - SQL like query language (HiveQL)
    - A subset of SQL with many traditional features
    - It is possible to embedded MR script in HiveQL
  - Queries are compiled into MR jobs that are executed on Hadoop.

HIVE's Data Model

- Data is organised in (non-NF1) tables
  - Rows with multiple columns of pre-defined types
  - Primitive data types
    - Integers (8, 16, 32 and 64 bits, all signed) (TINYINT – BIGINT)
    - Floating Points (single or double precision)
    - String
    - Boolean
  - Complex Data types
    - Associative arrays (map<key-type, value-type>)
    - Lists (list<element-type>)
    - Structs (field-name: type1, …)
    - Nested types
- There is also some control on the physical data layer with *partitions* and *buckets*
  - See also: https://cwiki.apache.org/Hive/tutorial.html

HIVE’s Query Language (HiveQL)

- Based on SQL
  - SELECT … FROM … WHERE … GROUP BY … ORDER BY … LIMIT
  - Including aggregates and grouping
  - ANSI JOIN syntax
- Example:
  - **SELECT** t1.a1 as c1, t2.b1 as c2
    FROM t1 **JOIN** t2 **ON** (t1.a2 = t2.b2)
- Restrictions:
  - Meant for DHW queries, so no per-row INSERT, DELETE, UPDATE
  - Only equi-joins?
- Plus Extensions: e.g. UDFs

HiveQL Example 1: Simple Query

- Table creation & Data loading

```
CREATE TABLE photos(user STRING, taken STRING, place_id STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

LOAD DATA INPATH '/user/zhouy/n05.txt' OVERWRITE INTO photos;
```

- Simple Query

```
SELECT * FROM photos where place_id = 'xPOpvDeYBJkdlw';
```

For this simple query, just a mapper phase is used.
No reducer is required.
Actually, two mappers will be generated because the table data (n05.txt) is split into two blocks.
The query does not specify where to store the output, it is displayed on the console.

```
Map:
        Input: (user,taken,place_id)
        Output: (user,taken,place_id) if place_id == 'xPOpvDeYBJkdlw'
        otherwise emit nothing
```

HiveQL Example 2: JOIN Query

- Join & writing data into file system from query
  - ANSI syntax: *R* JOIN *S* ON (*condition*)
  - Outer joins supported
  - Reduce side join

Typically we write query results not out to screen, but either to a file or a result table

```
INSERT OVERWRITE  DIRECTORY 'hiveout'
  SELECT ph.user, ph.taken, pl.place_name
    FROM photos ph LEFT OUTER JOIN places pl ON(ph.place_id=pl.place_id)
```

Reduce input groups	0	2,803,894	2,803,894
Combine output records	0	0	0
Map input records	2,803,894	0	2,803,894
Reduce shuffle bytes	0	92,944,673	92,944,673
Reduce output records	0	0	0
Spilled Records	5,607,788	2,803,894	8,411,682
Map output bytes	171,872,114	0	171,872,114
Map input bytes	158,803,223	0	158,803,223
Map output records	2,803,894	0	2,803,894
Combine input records	0	0	0
Reduce input records	0	2,803,894	2,803,894

For this simple join the execution time is similar as hand-coded map reduce jobs.

The process is also similar

There are 3 mappers and 1 reducers

The output can be written to a table as well

Example 3: Grouping and Aggregation

- Aggregator
  - On our example data set: two mappers, one reducer

```
INSERT OVERWRITE DIRECTORY 'hive/aggregator'
      SELECT ph.place_id, COUNT(DISTINCT ph.user)
        FROM photos ph
       GROUP BY ph.place_id;
```

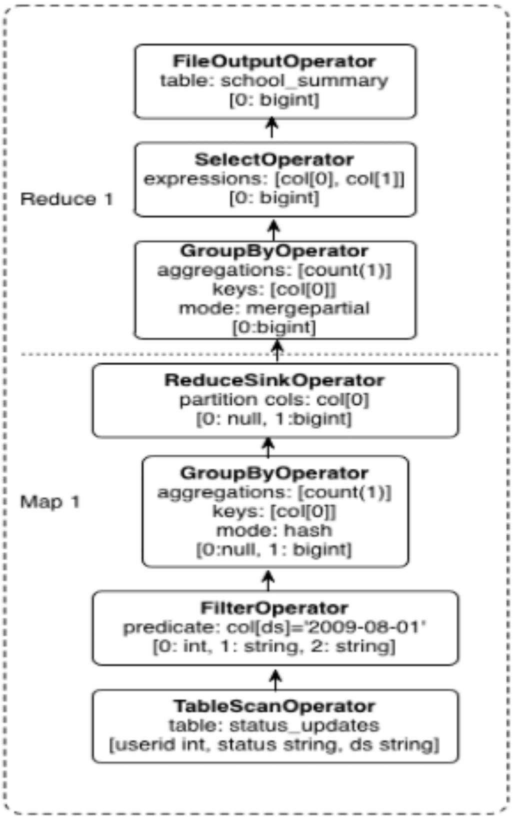
```
Map:
      V:(user,taken,place_id) -> K:place_id,V: user
Reduce:
      k:place_id, v:(user1,user2,user1,…) -> k:place_id, v:count
```

A combiner would be useful here, a combiner uses the same interface as the Reducer

```
Reduce:
      k:place_id, v: (user1, user1, user3, user2,..) ->
                                   k:place_id, v:user1;
                                   k:place_id, v:user3;
                                   k:place_id, v:user2;
```

Example Query (Aggregation)

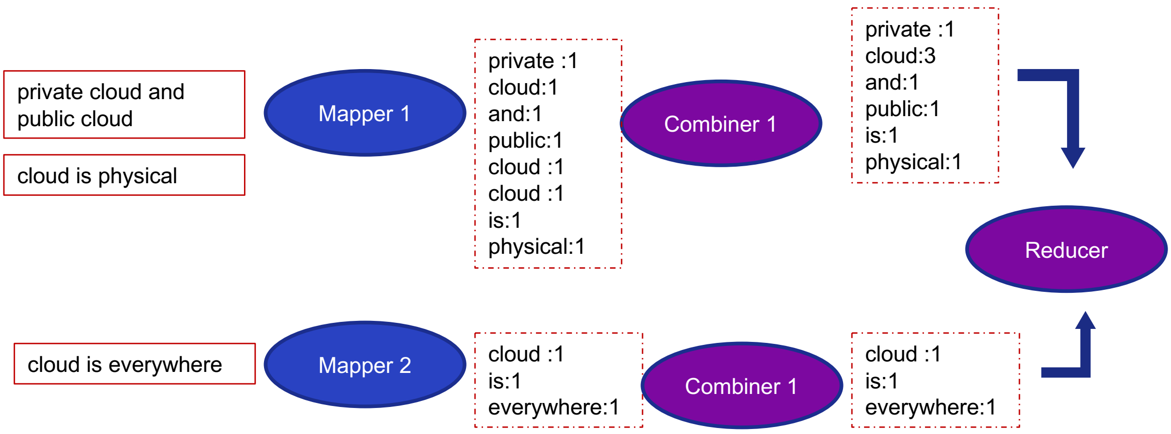
- Figure out total number of status_updates in a given day
  - `SELECT COUNT(1) FROM status_updates WHERE ds = '2009-08-01'`



Slide from Facebook presentation at ICDE 2010 (http://www.slideshare.net/ragho/hive-icde-2010)

Combiner Functions

- Combiner functions run on the mapper side; the idea is to minimize the data transferred between map and reduce tasks
- Recall the combiner function in the word count example
  - It is the same as the Reducer



Mapper output records **Actual shuffled records**

Why can't we use the Reducer as the combiner in the aggregator example?

HiveQL Example 4: Subqueries

- Subqueries involving join and aggregator are allowed at the FROM clause
  - They need to be named
  - Example:

```
INSERT OVERWRITE DIRECTORY 'hive/join_group'
  SELECT pl.place_name, agg.c
    FROM places pl LEFT OUTER JOIN (
              SELECT ph.place_id as pid,
                     COUNT(DISTINCT ph.user) AS c
                FROM photos ph
               GROUP BY ph.place_id) agg
         ON (pl.place_id = agg.pid);
```

  - Two MapReduce jobs are generated an aggregator one and a join one

Hive Architecture

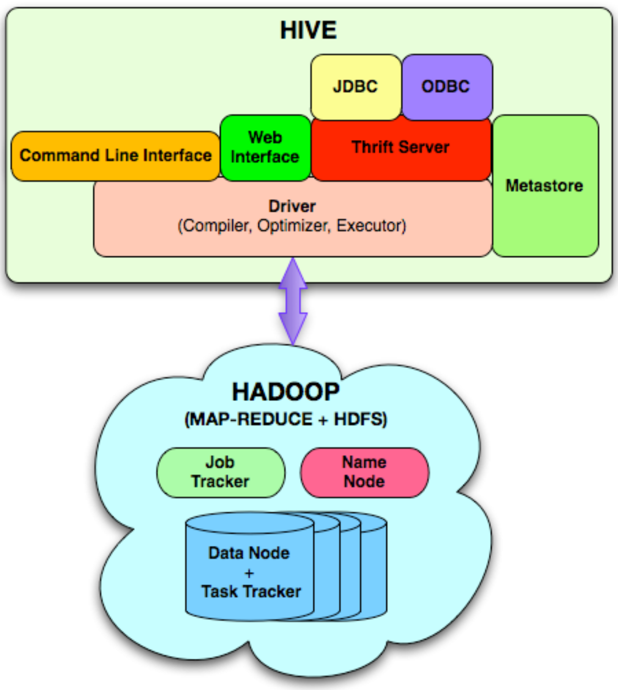


Diagram from the HIVE 2011 paper

Hive Architecture

- **Metastore**: stores system catalog
- **Driver**: manages life cycle of HiveQL query as it moves thru' HIVE; also manages session handle and session statistics
- **Query compiler**: Compiles HiveQL into a directed acyclic graph of map/reduce tasks
  - According to ICDE paper, inlcudes:
  - Rule-based Query Optimizer
    - E.g. predicate push-down or partition pruning (if condition is not satisfied)
- **Execution engines**: The component executes the tasks in proper dependency order; interacts with Hadoop
- **HiveServer**: provides Thrift interface and JDBC/ODBC for integrating other applications.
- **Client components**: CLI, web interface, jdbc/odbc inteface
- Extensibility interface include SerDe, User Defined Functions and User Defined Aggregate Function.

The Performance Gap

- For simple queries, HIVE performance is comparable with hand-coded MR jobs
- The execution time is much longer for complex queries
  - HiveQL allows for arbitrary number of embedded subqueries.
  - They are converted to MapReduce jobs respectively
  - The simple conversion may involve many unnecessary data scan and transfer.

Slides 44 - 53 are based on Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang, **YSmart: Yet Another SQL-to-MapReduce Translator**, In Proceedings of 31st International Conference on Distributed Computing Systems (ICDCS 2011)

Example of a Complex Query

- Input data: clickstream table of the format
  - *CLICKS(uid* int, *pid* int, *cid* int, *ts* timestamp*).*
- Query (Q-CSA) wants to find out
  - *What is the average number of pages a user visits between a page in category X and a page in category Y?*

```
SELECT avg(pageview_count) FROM
(SELECT c.uid,mp.ts1,(count(*)-2) AS pageview_count
 FROM clicks AS c,
   (SELECT uid,max(ts1) AS ts1,ts2
    FROM (SELECT c1.uid,c1.ts AS ts1,min(c2.ts) AS ts2
          FROM clicks AS c1,clicks AS c2
          WHERE c1.uid = c2.uid AND c1.ts < c2.ts
                AND c1.cid = X AND c2.cid = Y
          GROUP BY c1.uid,ts1) AS cp
   GROUP BY uid,ts2) AS mp
 WHERE c.uid=mp.uid AND c.ts>=mp.ts1 AND c.ts<=mp.ts2
 GROUP BY c.uid,mp.ts1) AS pageview_counts;
```

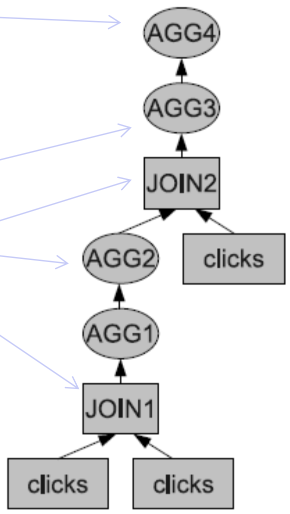
Fig. 1. The SQL statement for the clickstream analysis query (Q-CSA).

Figure 1 in YSmart paper

Query Execution and MR Jobs


```
SELECT avg(pageview_count) FROM
(SELECT c.uid,mp.ts1,(count(*)-2) AS pageview_count
 FROM clicks AS c,
   (SELECT uid,max(ts1) AS ts1,ts2
    FROM (SELECT c1.uid,c1.ts AS ts1,min(c2.ts) AS ts2
          FROM clicks AS c1,clicks AS c2
          WHERE c1.uid = c2.uid AND c1.ts < c2.ts
                AND c1.cid = X AND c2.cid = Y
          GROUP BY c1.uid,ts1) AS cp
   GROUP BY uid,ts2) AS mp
 WHERE c.uid=mp.uid AND c.ts>=mp.ts1 AND c.ts<=mp.ts2
 GROUP BY c.uid,mp.ts1) AS pageview_counts;
```

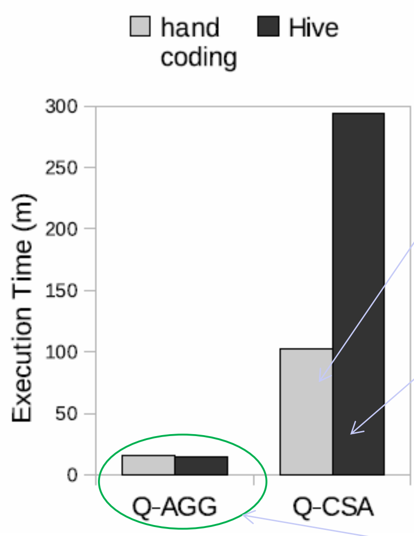
Fig. 1. The SQL statement for the clickstream analysis query (Q-CSA).



(a) Q-CSA plan.

Figure 1 and 2 in YSmart paper

HIVE vs. Hand-coded MR Program



(b) Execution times.

Hand-coded MR program only requires 2 jobs

Automatically generated MR code uses 6 jobs

Simple aggregation query, like the one in slide 37; Hive uses an optimized execution strategy for aggregations by maintaining an internal hash-aggregate map in the map phase of a job

Figure 2 in YSmart paper

Limitations of MapReduce for Complex Queries

- A one-operation-to-one-job translation is constrained by the structure and implementation of MapReduce in two ways
  - MapReduce requires materialization of intermediate results on local disks
  - the run-time system (e.g., Hadoop) is not aware whether concurrent jobs are correlated, thus it does not provide any mechanism to support intermediate data reusing between concurrent jobs
- Intra-query correlation
  - Many workload contains queries on <u>multiple occurrences</u> of the <u>same</u> table, including self-joins.
  - considering intra-query correlations, SQL-to-MapReduce translations and executions can be automatically optimized to significantly improve performance through minimizing computation and I/O operations by merging correlated query operations

Optimization Solution

- Identifying various correlations
  - **Input** correlation
    - Multiple nodes have input correlation if their input relation sets are not disjoint
  - **Transit** correlation
    - Multiple nodes have transit correlation (TC) if they have not only input correlation, but also the same partition key;
  - **Job flow** correlation
    - A node has job flow correlation (JFC) with one of its child nodes if it has the same partition key as that child node
- The idea is to figure out if jobs can be merged

Merge Example

```
SELECT sum(l_extendedprice) / 7.0 AS avg_yearly
FROM (SELECT l_partkey, 0.2* avg(l_quantity) AS t1
      FROM   lineitem
      GROUP BY l_partkey) AS inner,
     (SELECT l_partkey,l_quantity,l_extendedprice
      FROM   lineitem, part
      WHERE  p_partkey = l_partkey) AS outer
WHERE  outer.l_partkey = inner.l_partkey;
  AND  outer.l_quantity < inner.t1;
```

Fig. 3. A variation of TPC-H Q17.

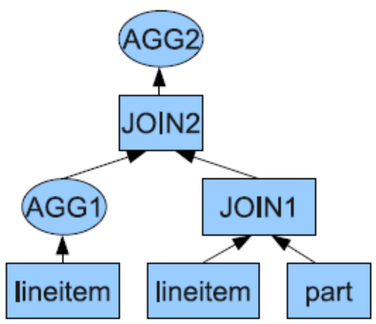


Fig. 4. The query plan tree for Q17.

Figure 3 and 4 in YSmart paper

Direct SQL – MapReduce Translation

```
Job1: generate inner by group/agg on lineitem
Map:
 lineitem -> (k:l_partkey, v:l_quantity)
Reduce:
 calculate (0.2*avg(l_quantity)) for each (l_partkey)

Job2: generate outer by join lineitem and part
Map:
 lineitem -> (k: l_partkey,
                 v:(l_quantity,l_extendedprice))
 part -> (k:p_partkey,v:null)
Reduce:
 join with the same partition (l_partkey=p_partkey)

Job3: join outer and inner
Map:
 outer-> (k:l_partkey, v:(l_quantity,l_extendedprice))
 inner-> (k:l_partkey, v:(0.2*avg(l_quantity)))
Reduce:
 join with the same partition of l_partkey
```

Fig. 5. A chain of jobs for the plan in Fig. 4. (We ignore the fourth job for evaluating the final aggregation AGG2)

Improved Program by Exploiting Correlation

```
Job1: generate both inner and outer,
      and then join them
Map:
 lineitem -> (k: l_partkey,
                 v:(l_quantity,l_extendedprice))
 part -> (k:p_partkey,v:null)
Reduce:
 get inner: aggregate l_quantity for each (l_partkey)
 get outer: join with (l_partkey=p_partkey)
 join inner and outer
```

Evaluation Result

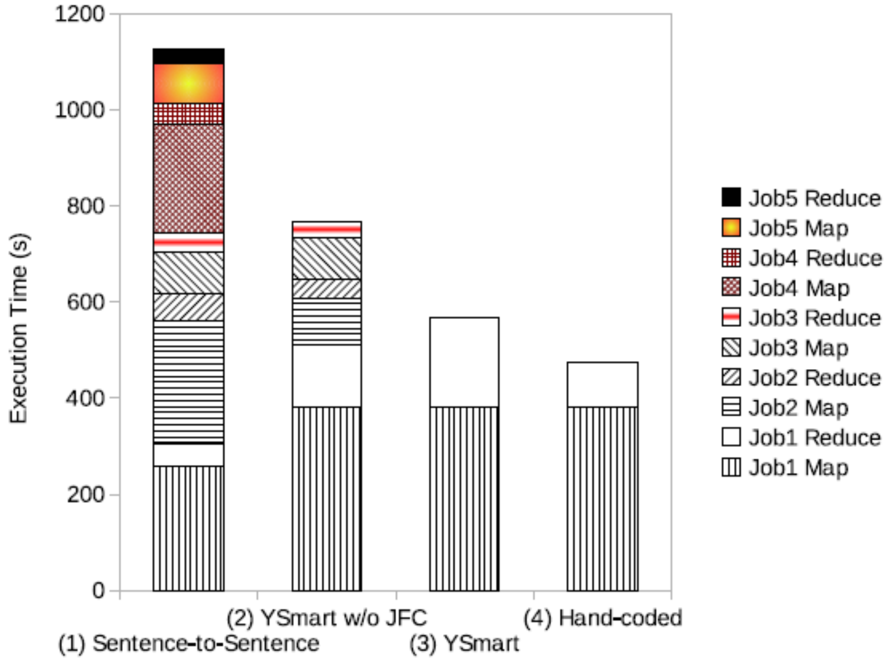
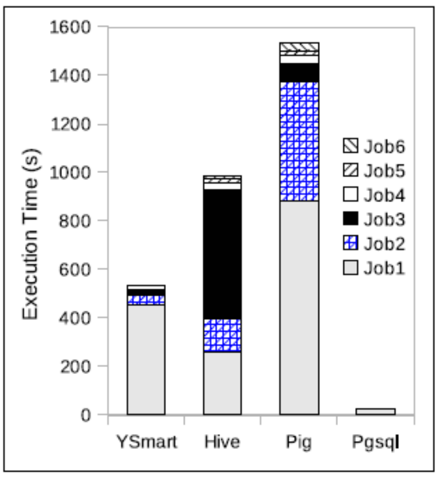


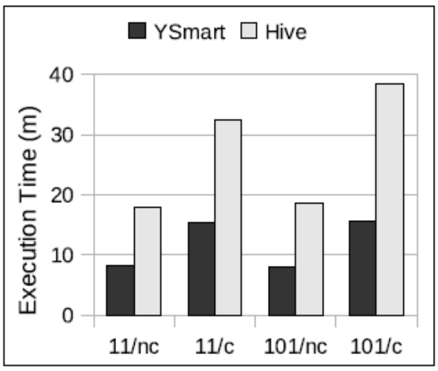
Fig. 9. Breakdown of job finishing times of Q21

Figure 9 in YSmart paper

Evaluation Results (cont'd)



(b) Q18



(b) Q18

Figure 10 and 11 in YSmart paper

Tenzing

Google's Tenzing [VLDB2011]

- SQL Layer on top of MapReduce
- Authors claim almost complete SQL Syntax compatibility
  - Filtering, joins, aggregation,…
  - Incl. many analytical constructs
  - Such as SQL window queries
    and SQL OLAP extensions (ROLLUP() and CUBE()
- Several optimizations for better efficiency and low latency
  - E.g. pool of worker processes always available to reduce startup-costs for new Map/Reduce tasks
  - Hash-based aggregation which avoids sorting between map() and reduce()
  - Cost-based query optimizer (no details published)

Tenzing Example: Rank()

- Tenzing Query:

  SELECT dept, emp, salary,
  RANK() OVER
  (PARTITION BY dept
  ORDER BY salary DESC
  FROM Employee;

- Map/Reduce Pseudo-Code

```
Mapper::Map(in) {
  // From the mapper, we output the partitioning key of
  // the analytic function as the key, and the ordering
  // key and other information as value.
  OutputToReducerWithSort(
    key = in.dept, value = {in.emp, in.salary})
}

Reducer::Reduce(key, values) {
  // Reducer receives all values with the same partitioning
  // key. The list is then sorted on the ordering key for
  // the analytic function.
  sort(values on value.salary)
  // For simple analytic function such as RANK, it is
      enough
  // to just print out the results once sorted.
  for (value in values) {
    print key, value.emp, value.salary, i
  }
}
```


Tenzing: Hash-Based Aggregation

- Tenzing Query:

  SELECT dept_id, COUNT(1)
  FROM Employee
  /∗+ HASH ∗/ **GROUP BY** 1;

- Map/Reduce Pseudo-Code
  =>

```
// In the mapper startup, initialize a hash table with dept_id
// as key and count as value.
Mapper::Start() {
  dept_hash = new Hashtable()
}

// For each row, increment the count for the corresponding
// dept_id.
Mapper::Map(in) {
  dept_hash[in.dept_id] ++
}

// At the end of the mapping phase, flush the hash table to
// the reducer, without sorting.
Mapper::Finish() {
 for (dept_id in dept_hash) {
   OutputToReducerNoSort(
     key = dept_id, value = dept_hash[dept_id])
 }
}

// Similarly, in the reducer, initialize a hash table
Reducer::Start() {
  dept_hash = new Hashtable()
}

// Each Reduce call receives a dept_id and a count from
// the map output. Increment the corresponding entry in
// the hash table.
Reducer::Reduce(key, value) {
  dept_hash[key] += value
}

// At the end of the reduce phase, flush out the
// aggregated results.
Reducer::Finish() {
  for (dept_id in dept_hash) {
    print dept_id, dept_hash[dept_id]
  }
}
```

Potential Problems with Data Analysis based on Map-Reduce

- Latency
  - In the order of minutes
  - Because of start-up costs and scheduling of the Map/reduce tasks
  - Even Google admits to be still in multiple seconds range
- SQL Compatibility
  - Low
- Efficiency
  - No good query optimization techniques
  - Materialization rather than pipelining during execution

Is this always a Winner?

- Example: Simple Scalability benchmark published by Google in their VLDB2011 Tenzing Paper
  - Query:
    SELECT a, SUM(b)
      FROM T
     WHERE c = k
     GROUP BY a
  - Performance:

Workers	Runtime	Rows/Worker/sec
100	188.74s	16.74
500	36.12s	17.49
1000	19.57s	16.14

M/R vs. RDBMS

4 different SQL queries (multi-way joins, aggregation, grouping)

Query	DBMS-X (s)	Tenzing (s)	Change
#2	129	93	39% faster
#4	70	69	1.4% faster
#1	155	213	38% slower
#3	9	28	3.1 times slower

Table 2: Tenzing versus DBMS-X.

Queries for Previous Example

A.1 Query 1

```
SELECT DISTINCT dim2.dim1_id
FROM FactTable1_XXB stats
INNER JOIN DimensionTable1_XXXM dim1
  USING (dimension1_id)
INNER JOIN DimensionTable2_XXM dim2
  USING (dimension2_id)
WHERE dim2.attr BETWEEN A and B
AND stats.date_id > some_date_id
AND dim1.attr IN (Val1, Val2, Val3, ...);
```

A.3 Query 3

```
SELECT attr4 FROM (
  SELECT dim4.attr4, COUNT(*) AS dup_count
  FROM DimensionTable4_XXM dim4
  JOIN DimensionTable5_XXM dim5
    USING (dimension4_id)
  WHERE dim4.attr1 BETWEEN Val1 and Val2
  AND dim5.attr2 IN (Val3, Val4)
  GROUP BY 1
  HAVING dup_count = 1) x;
```

A.2 Query 2

```
SELECT
  dim1.attr1, dates.finance_week_id,
  SUM(FN1(dim3.attr3, stats.measure1)),
  SUM(FN2(dim3.attr4, stats.measure2)),

FROM FactTable1_XXB stats
INNER JOIN DimensionTable1_XXXM dim1
  USING (dimension1_id)
INNER JOIN DatesDim dates
  USING (date_id)
INNER JOIN DimensionTable3_XXXK dim3
  USING (dimension3_id)
WHERE <fact date range>
GROUP BY 1, 2;
```

A.4 Query4

```
SELECT attr1, measure1 / measure2
FROM (
  SELECT
    attr1,
    SUM(FN1(attr2, attr3, attr4)) measure1,
    SUM(FN2(attr5, attr6, attr7)) measure2,
  FROM DimensionTable6_XXM
  WHERE FN3(attr8) AND FN4(attr9)
  GROUP BY 1
) v;
```

Summary

- **MapReduce too low-level for some users**
  - Requires functional coding skills which are not available neither to many software developers, nor useful for non-IT users
- **Several Approaches to Data Analytics on top of M-R**
  - Higher-level languages that can get automatically mapped to a series of Map-Reduce jobs
  - Some are data-flow driven
    - E.g. Pig Latin from Apache (org. Yahoo!)
    - Targeting more high-level programmers
  - Some are more SQL based
    - HIVE and Tenzing
    - Targeting data analysts who like SQL

References

- Facebook
  - Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu and Raghotham Mur: Hive – A Petabyte Scale Data Warehouse Using Hadoop. In *Proceedings of ICDE* 2010.
  - Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang: YSmart: Yet Another SQL-to-MapReduce Translator, In *Proceedings of ICDCS 2011*.
- Google
  - B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, W. Lychagina, Y. Kwon and M. Wong: Tenzing: A SQL Implementation on the MapReduce Framework. In *Proceedings of VLDB2011*.
  - Rob Pike et all: Interpreting the Data: Parallel Analysis with Sawzall. Scientific Programming, 2005
- Yahoo!
  - Christopher Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. SIGMOD 2008.