Data Warehousing

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

Summer 2019

Part VII

MapReduce *et al.*

Scaling Up Data Warehouse Systems

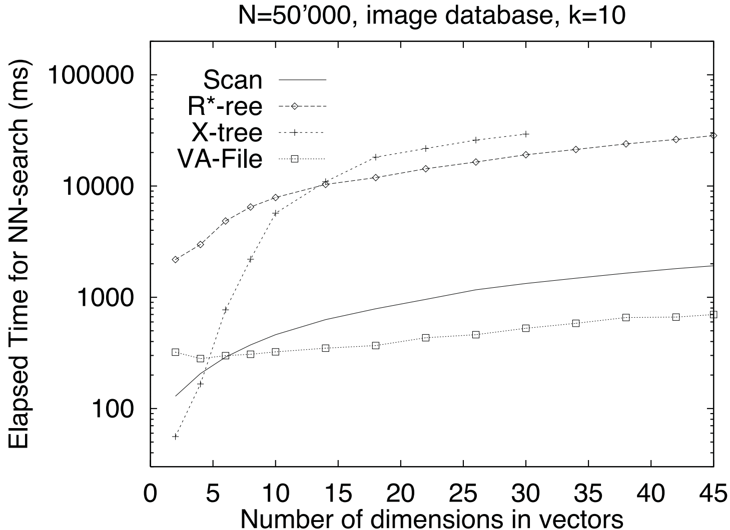
Growing **expectations** toward Data Warehouses:

- increasing **data volumes** ("Big Data")
- increasing **complexity of analyses**

Problems:

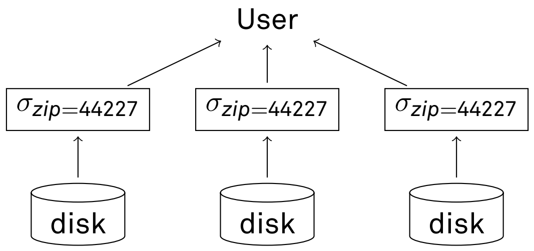
- OLAP queries are **multi-dimensional** queries
  - → "Curse of Dimensionality:" indexes become ineffective
  - → Indexes can't help to fight growing query complexity
  - → Workloads become **scan heavy.**
- Scaling up a server becomes **expensive**

Curse of Dimensionality



Parallel Query Evaluation

Scans can be **parallelized**, however:



→ parallel hardware (*e.g.*, graphics processors)

→ cluster systems

E.g., Jedox OLAP Accelerator
(uses NVIDIA Tesla GPUs):

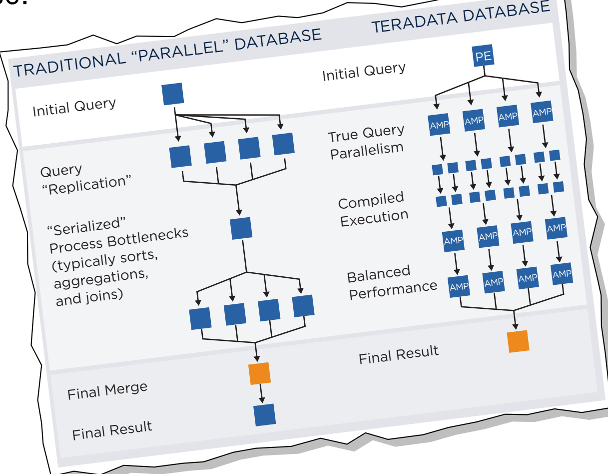
source:
Jedox White Paper

In-GPU-Memory

Moderne GPU-Module bestehen aus tausenden Prozessoreinheiten und zeigen gerade in komplexen und parallelisierbaren Berechnungen ihren Performanzvorteil gegenüber CPUs. Die parallele Verarbeitung erhöht die Geschwindigkeit von Zugriffen und Datenanalysen signifikant, besonders wenn es sich um Berechnungen mit konsolidierten Zellen im OLAP-Datenwürfel handelt.

Um zeitintensiven Datentransfer vom Hauptspeicher auf die GPU zu minimieren, setzt Jedox auf innovative „In-GPU-Memory"-Technologie: Der Jedox GPU Accelerator hält die Zelldaten der Würfel vollständig im GPU-Speicher, so dass lediglich Anfragen und Ergebnisse zwischen CPU und GPU übertragen werden müssen. Für den Nutzer laufen die Anwendungen dadurch deutlich schneller und alle wichtigen Daten werden sofort bereitgestellt. Zudem können auch mehrere GPUs verwendet werden, um so für noch kürzere Abfragezeiten und einen größeren GPU-Speicher für besonders große Datenwürfel zu sorgen.

E.g., Teradata Database:



source:
Teradata White Paper

Scalability Challenges

Challenges:

- **Robustness:**
  - More components → higher risk of failure
  - Failure of single component might take whole system off-line.
- **Scalability/Elasticity:**
  - Provision for peak load?
  - Use resources otherwise when DW not at peak load?
  - Add resources later (when business grows)?
- **Cost:**
  - (Reliable) large installations tend to become expensive.
    (There's a relatively small market for very large systems.)

Scalability in Web Search

Search engines have faced similar challenges very early.

Task: generate **inverted files**

data warehouses are cool

doc_1

cool guys distribute their data

doc_2

⟶

term	cnt	"posting list"
are	1	doc_1:3
cool	2	doc_1:4, doc_2:1
data	2	doc_1:1, doc_2:5
distribute	1	doc_2:3
guys	1	doc_2:2
their	1	doc_2:4
warehouses	1	doc_1:2

Inverted File Generation

Idea: Break up index generation into two parts:

1. For each document, extract terms.
2. Collect terms into groups and emit an index entry per group.

E.g.,

```
1  foreach document doc do
2      pos ← 1;
3      tokens ← parse(doc);
4      foreach w in tokens do
5          emit ⟨w, doc.id:pos⟩;
6          pos ← pos + 1;
7  collect ⟨w, [doc.id:pos ...]⟩ pairs;
```

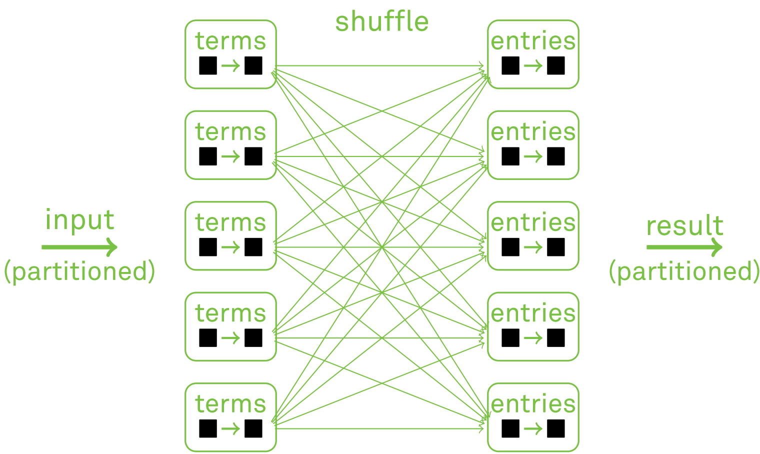
```
8  foreach ⟨w, [doc.id:pos ...]⟩
   do
9      count ← 0;
10     pList ← ();
11     foreach
        v ∈ [doc.id:pos ...] do
12         pList.append(v);
13         count ← count + 1;
14     emit ⟨key, count, pList⟩;
```

Inverted File Generation

Observations: (for parallel execution)

- For part 1, documents can be partitioned **arbitrarily** over nodes.
- For part 2, all postings of one term must be **collocated** on the same node (postings for different terms may be on different nodes).
- To establish collocation, data may have to be **moved** ("shuffled") across nodes.

Distributed Index Generation



Generalization ($\rightarrow$ "MapReduce")

The application pattern turns out to be highly versatile.

Only replace **foreach** bodies:

- **lines 2–6:** $f_1 :: \alpha \rightarrow [\langle \beta, \gamma \rangle]$ $\rightarrow$ "Mapper"
- **lines 8–14:** $f_2 :: \langle \beta, [\gamma] \rangle \rightarrow \delta$ $\rightarrow$ "Reducer"

Shuffling (line 7) **combines** $[\langle \beta, \gamma \rangle]$ ("list of key/value pairs") into a list of $\langle \beta, [\gamma] \rangle$ ("pairs of key and list of values").

$\rightarrow$ Shuffling (combining) is generic.

MapReduce[3] is a framework for distributed computing, where f_1 and f_2 can be instantiated by the user.

[3] Dean and Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI 2004*.

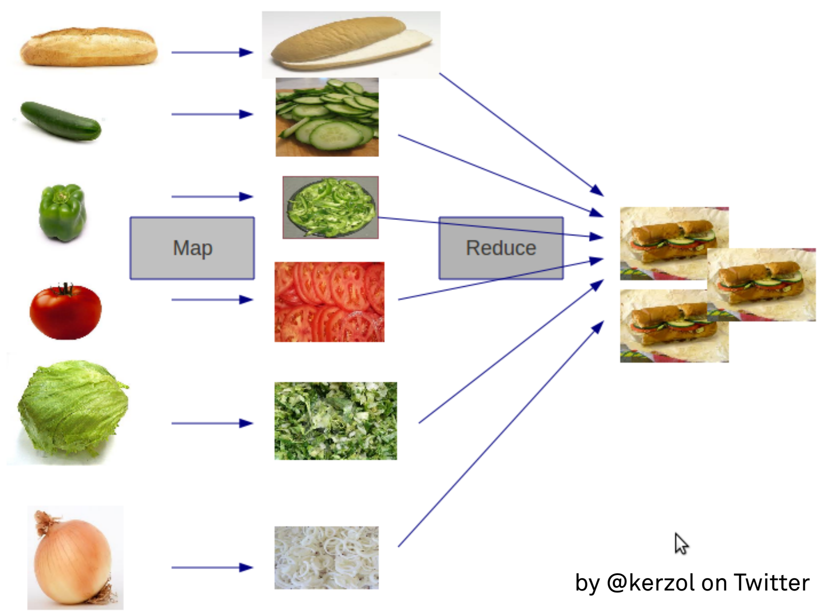
Example: Webserver Log File Analysis

E.g., Webserver log file analysis

Task: For each client IP, report total traffic (in bytes).

✎ **Mapper and Reducer implementations?**

MapReduce Illustrated



by @kerzol on Twitter

MapReduce

The MapReduce framework...

- ...decides on a number of Mappers and Reducers to instantiate,
- ...decides the partitioning of of data and computation,
- ...moves data as necessary and implements shuffling;

- ...considers cluster topology, system load, etc.,
- ...interfaces with a distributed file system ("Google File Syst.").

Apache Hadoop provides an open-source implementation of the MapReduce concept; also comes with the "Hadoop Distributed File System, HDFS."

So What?

"The idea seems straightforward. Why all the fuss?"

Remember the challenges we stated?

→ Risk of failures; elasticity; cost

MapReduce was designed for **large clusters** of **cheap machines**.

→ Think of thousands of machines.

→ Failures are **frequent** (and have to be dealt with).

→ **This** is why MapReduce has become popular.

Failure Tolerance?

Trick:

- Mapper and Reducer must be **pure** functions.
  - → Their output depends **only** on their input.
  - → **No** side effects.
- Computation can be done **anywhere**, **repeated** if necessary.

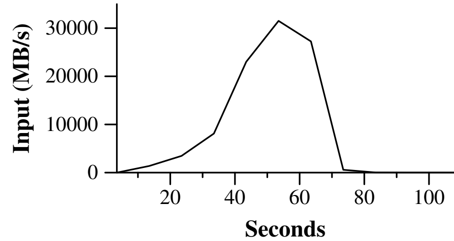
MapReduce runtime:

- Monitor job execution.
- Job does not finish within expected time?
  - → Restart on different node.
  - → Might end up processing a task unit twice → discard all results but one.
  - → Also used to improve performance (in case of "stragglers").

Performance: Grep

E.g., scan 10^{10} 100-byte words for three-character pattern.

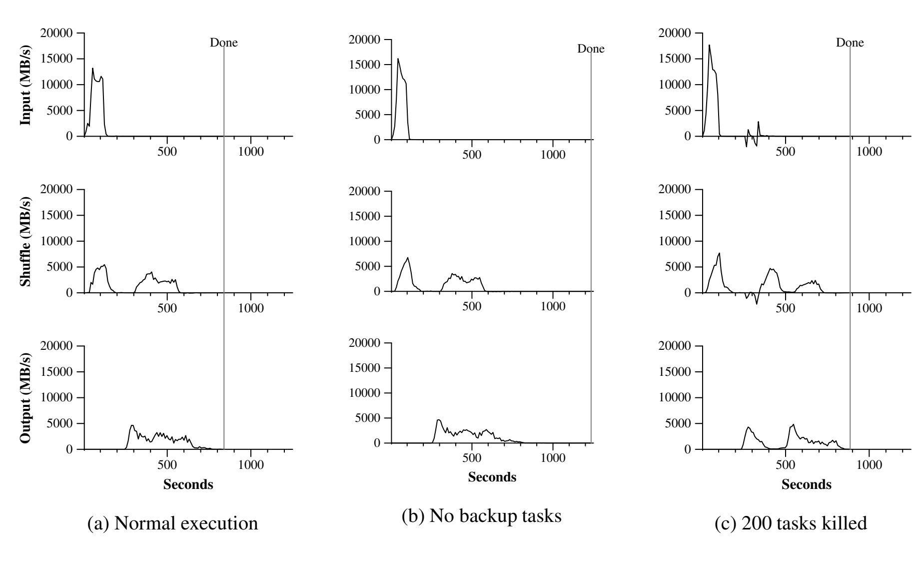
- 1800 machines
- each 2×2 GHz
- each 2×160 GB IDE HDD
- Gigabit Ethernet
- paper from 2004



→ Leverage **aggregate disk bandwidth**.

→ This is what we need for OLAP, too.

Performance: Sort



MapReduce for Data Warehousing

MapReduce is **not** a database.

→ No tables, tuples, rows, schemas, indexes, etc.

Rather, MapReduce is based on **files**.

→ Typically kept in a **distributed file system**.

This is **unfortunate**:

- No schema information to optimize, validate, etc.
- No indexes (or other means to improve physical representation).

This is **good**:

- Start analyzing immediately; don't wait for index creation, etc.
- May ease ad-hoc analyses.

Beyond the Basic Idea

While the original MapReduce is proprietary to Google, Hadoop is widely used in industry and research.

- Java-based
  - → Can run on heterogeneous platforms, cloud systems, etc.
- Integration with other Apache technology
  - → Hadoop Distributed File System (HDFS), HBase, etc.
- Can hook into more functions than just Mapper and Reducer
  - → *e.g.*, pre-aggregate between map and shuffle
  - → modify partitioning, etc.
- Many interfaces Hadoop ↔ database/data warehouse

Hadoop and Petabyte Sort Benchmark

Challenge: sort 1 TB of 100-byte records.

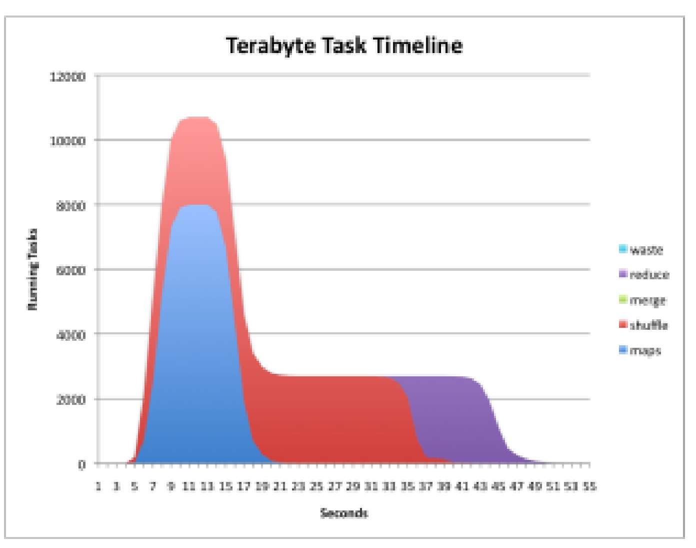
Hardware:

- 3800 nodes, $2 \times 4 \times 2.5$ GHz per node
- 4 SATA disks, 8 GB RAM per node

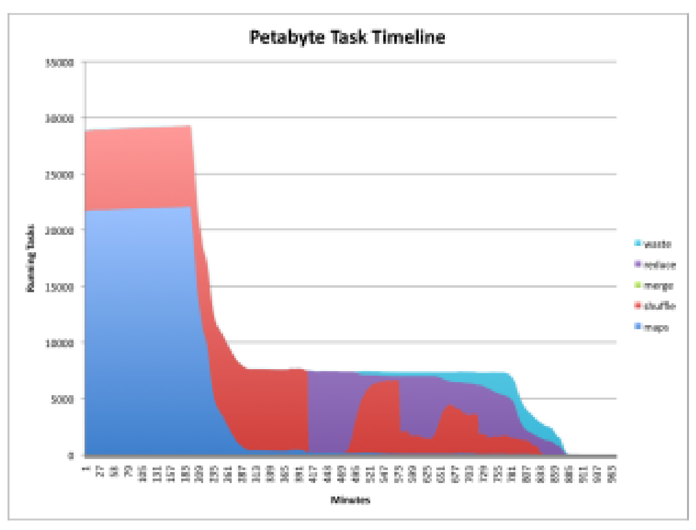
Results:

GBytes	Nodes	Maps	Reduces	Repl.	Time
500	1406	8000	2600	1	59 sec
1,000	1460	8000	2700	1	62 sec
100,000	3452	190,000	10,000	2	173 min
1,000,000	3658	80,000	20,000	2	975 min

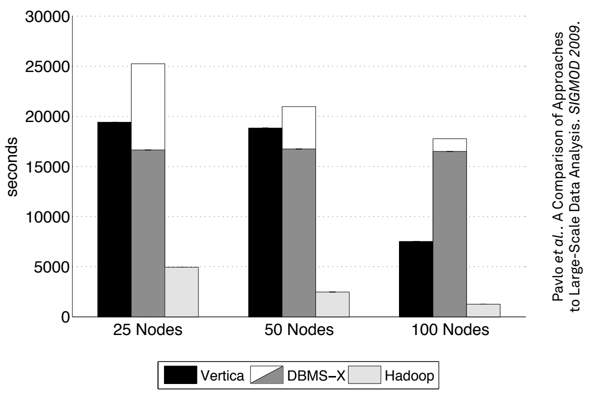
Hadoop and Petabyte Sort Benchmark



Hadoop and Petabyte Sort Benchmark

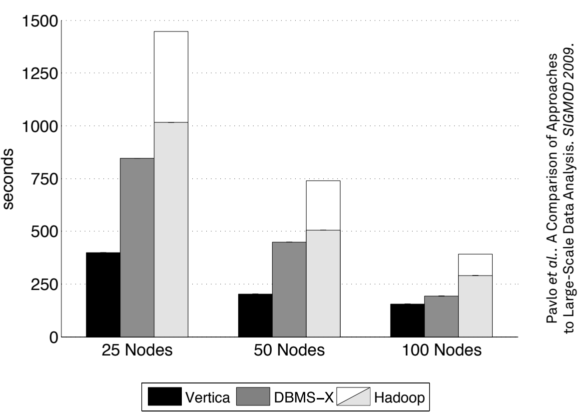


MapReduce ↔ Databases: Load Times



→ Schema and physical data organization make loading slower on the databases.

MapReduce ↔ Databases: Grep Benchmark



→ MapReduce leaves result as collection of files; collecting into single result costs addl. time.

MapReduce ↔ Databases: Aggregation

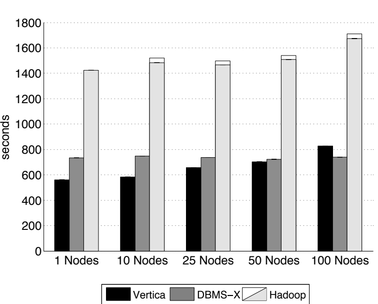


Figure 7: Aggregation Task Results (2.5 million Groups)

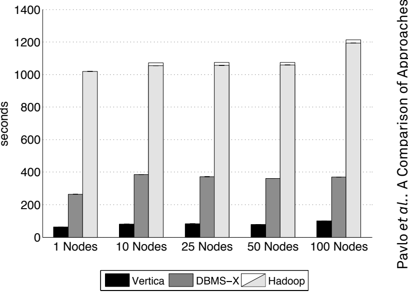
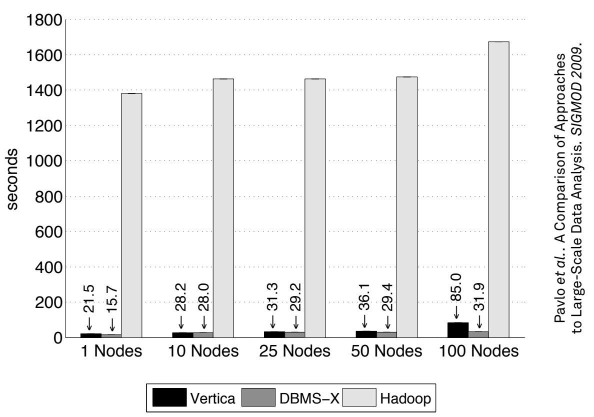


Figure 8: Aggregation Task Results (2,000 Groups)

Pavlo *et al.*. A Comparison of Approaches to Large-Scale Data Analysis. *SIGMOD 2009.*

→ Databases limited by communication cost, which is lower for smaller group counts.

MapReduce ↔ Databases: Join



Pavlo *et al.*. A Comparison of Approaches to Large-Scale Data Analysis. *SIGMOD 2009*.

→ Joins are rather complex to formulate in MapReduce.

→ Repartitioning incurs high communication overhead.

→ Joins can be accelerated using indexes.

MapReduce $\leftrightarrow$ Databases

Persistent data $\leftrightarrow$ data read ad-hoc:

- Overhead for schema design, loading, indexing, etc.
  - $\rightarrow$ Cost might amortize only after several queries/analyses.
- Databases feature support for transactions.
  - $\rightarrow$ Not needed for read-only workloads.

Language: SQL $\leftrightarrow$ Java/C++/...:

- Write a new MapReduce program for each and every analysis?
- User-defined functionality in SQL?
  - $\rightarrow$ *E.g.*, similarity measures, statistics functions, etc.
- Debug SQL or MapReduce job?

Is there a good middle ground?

Apache Pig

Idea:

- Data processing language that sits in-between SQL and MapReduce.
  - → Declarative ("SQL-like"; $\leadsto$ allow for optimization, easy re-use and maintenance)
  - → Procedural-style, rich data model ($\leadsto$ programmers feel comfortable)
- Pig programs are **compiled into MapReduce (Hadoop) jobs**.

Pig Latin Example

```
S = LOAD 'sailors.csv' USING PigStorage(',')
      AS (sid:int, name:chararray, rating:int, age:int);
B = LOAD 'boats.csv' USING PigStorage(',')
      AS (bid:int, name:chararray, color:chararray);
R = LOAD 'reserves.csv' USING PigStorage(',')
      AS (sid:int, bid:int, day:chararray);

-- SELECT S.sid, R.day
--   FROM Sailors AS S, Reserves AS R
--  WHERE S.sid = R.sid AND R.bid = 101

A = FILTER R BY (bid == 101);
B = JOIN S BY sid, A by sid;
X = FOREACH B GENERATE S::sid, A::day AS day;

STORE X into 'result.csv';
```

schema on-the-fly

programming style:
sequence of assignments
⤳ data flow

Pig Latin Data Model

Pig Latin features a fairly rich data model:

atoms:

→ *e.g.*, `'foo'`, 42

tuples: sequence of *fields* of any data type

→ *e.g.*, (`'foo'`, 42)

→ access by *field name* or *position*, tuples can be nested

bag: collection of tuples (possibly with duplicates)

→ *e.g.*, $\left\{ \begin{array}{c} (\texttt{'foo'}, 42) \\ (17, (\texttt{'hello'}, \texttt{'world'})) \end{array} \right\}$

map: collection of *key* → *value* mappings

→ *e.g.*, $\left[\begin{array}{c} \texttt{'fan of'} \to \left\{ \begin{array}{c} (\texttt{'lakers'}) \\ (\texttt{'iPod'}) \end{array} \right\} \\ \text{age} \to 20 \end{array} \right]$

Pig Latin Data Model

- Pig Latin's data types can be **arbitrarily nested**[4]
- Contrast to 1NF data model in relational databases
  - → Avoid **joins**, which MapReduce can't do too well.
  - → Allow for **sound data model**, including grouping, etc.
  - → Easier integration with **user-defined functions**

[4] Keys for map types must be atomic, though (for efficiency reasons).

Pig Latin Operators: FILTER

```
kids = FILTER users BY (age < 18);
```

- → Comparison operators: `==`, `eq`, `!=`, `neq`, `AND`, ...
- → Can use **user-defined functions** arbitrarily.

✎ **Implementation in MapReduce?**

Pig Latin Operators: FOREACH

```
FOREACH Sailors GENERATE
  sid AS sailorId,
  name AS sailorName,
  ( rating, age ) AS sailorInfo;
```

- → Apply some processing (*e.g.*, item re-structuring) to every item of a data set (⇝ projection in Relational Algebra)
- → **No loop dependence!** → parallel execution
  (XQuery's `FLWOR` expressions provide a similar form of iteration.)

Pig Latin Operators: GROUP

```
sales_by_cust = GROUP sales BY customerName;
```

→ returns a bag ("relation") with two fields: group key and bag of tuples with that key value.

- First field is named `group`
- Second field is named by variable ("alias" in Pig terminology) used in the `GROUP` statement (here: `sales`)

✎ **Implementation in MapReduce?**

Pig Latin Operators: COGROUP

Group items from **multiple** data sets:

```
O = LOAD 'owner.csv' USING PigStorage(',')
      AS (owner:chararray, pet:chararray);
```

→ {(Alice, turtle), (Alice, goldfish), (Alice, cat), (Bob, dog), (Bob, cat)}

```
F = LOAD 'friend.csv' USING PigStorage(',')
      AS (person:chararray, friend:chararray);
```

→ {(Cindy, Alice), (Mark, Alice), (Paul, Bob), (Paul, Jane)}

```
X = COGROUP O BY owner, F BY friend;
```

$$\rightarrow \left\{ \begin{array}{l} \left(\texttt{Alice}, \left\{ \begin{array}{c} (\texttt{Alice}, \texttt{turtle}) \\ (\texttt{Alice}, \texttt{goldfish}) \\ (\texttt{Alice}, \texttt{cat}) \end{array} \right\}, \left\{ \begin{array}{c} (\texttt{Cindy}, \texttt{Alice}) \\ (\texttt{Mark}, \texttt{Alice}) \end{array} \right\} \right) \\ \left(\texttt{Bob}, \left\{ \begin{array}{c} (\texttt{Bob}, \texttt{dog}) \\ (\texttt{Bob}, \texttt{cat}) \end{array} \right\}, \left\{ (\texttt{Paul}, \texttt{Bob}) \right\} \right) \\ \left(\texttt{Jane}, \{\}, \left\{ (\texttt{Paul}, \texttt{Jane}) \right\} \right) \end{array} \right\}$$

Pig Latin Operators: JOIN

```
join_result = JOIN results BY queryString,
                   revenue BY queryString;
```

→ Equi-joins only.

✎ **Implementation in MapReduce?**

→ Cross product between fields 1 and 2 of COGROUP result.

```
temp = COGROUP results BY queryString;
               revenue BY queryString;

join_result = FOREACH temp GENERATE
                FLATTEN (results), FLATTEN (revenue);
```

Pig Latin: More Operators

Many additional operators ease common data analysis tasks, *e.g.*,

- `LOAD`/`STORE`
  (Not surprisingly, Pig works well together with HDFS.)
- `UNION`
- `CROSS`
- `ORDER`
- `DISTINCT`

Pig Latin: Debugging

Pig Latin was also designed with the development and analysis workflow in mind.

→ **Interactive** use of Pig ("grunt").

→ Can run Pig programs **locally** (without Hadoop).

→ Commands to examine expression results.

- `DUMP`: Write (intermediate) result to storage.
- `DESCRIBE`: Print schema of an (intermediate) result.
- `EXPLAIN`: Print execution plan.
- `ILLUSTRATE`: View step-by-step execution of a plan; show **representative examples** of (intermediate) result data.