Welcome to:

Unit 3 – Query languages for Hadoop

Unit objectives

After completing this unit, you should be able to:

- Understand overview of Jaql
- Understand basics of Jaql Language
- Understand Core Operators of Jaql Language
- Understand SQL Support for Jaql Language
- Understand MapReduce implementation in JAQL
- Understand Jaql I/O Systems

What is Jaql?

- Jaql: A JSON Query Language
- Query processing for *semi-structured* data
  - Represented using JSON
- Transparently exploits massive *parallelism*
  - Through the use of Apache Hadoop's Map-Reduce
- Modular and easily *extensible*
  - Native functions and modules allow packaging and re-use
  - Plug-in functions using your favorite programming language

JSON – JavaScript Object Notation

- It is a data-interchange format
  - Based on a subset of the JavaScript programming language
- Based upon two structures
  - A collection of name / value pairs
    - An object or record
  - A ordered list of values
    - An array

```
[
   {author: "David Baldacci", title: "The Camel Club",
   published: 2006},
   {author: "David Baldacci", title: "The Collectors",
   published: 2007},
   {author: "Rick Riordan", title: "The Red Pyramid",
   published: 2010},
   {author: "Rick Riordan", title: "The Throne of Fire",
   published: 2011,
       Reviews:[{rating: 10, reviewer: "Joe"},
                       {rating: 9, reviewer: "Sue"}]}
]
```

JSON format

- Objects are bounded by { and } and are separated by commas

```
{author: "David Baldacci", title: "The Camel Club", published: 2006},
{author: "David Baldacci", title: "The Collectors", published: 2007}
```

- Objects are grouped into arrays
  - Arrays are bounded by [and]

```
[ {author: "David Baldacci", title: "The Camel Club", published: 2006},
  {author: "David Baldacci", title: "The Collectors",       published: 2007} ]
```

- Objects are comprised of members
  - Members are made up of name / value pairs

```
author: "David Baldacci"
```

- Objects in an array can vary

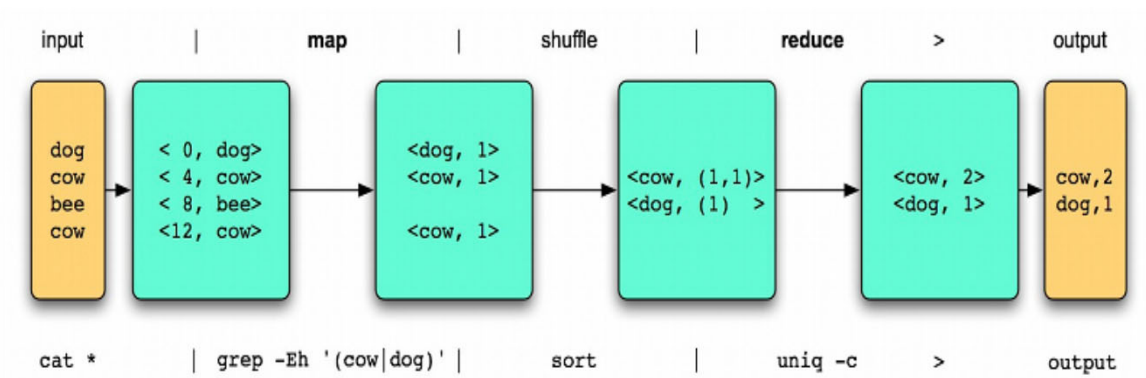
```
[{author: "Rick Riordan", title: "The Red Pyramid", published: 2010},
 {author: "Rick Riordan", title: "The Throne of Fire", published: 2011,
   reviews:[{rating: 10, reviewer: "Joe"},
            {rating: 9, reviewer: "Sue"}]}
]
```

Where does Jaql fit?

- Hive
  - Good for 'flat' data
  - Has UDF/UDA's
  - Uses familiar SQL syntax
- Pig
  - Better for more complex data
  - Has UDF/UDA's
  - Used for simple scripts
- Jaql
  - Best for very complex/nested data (e.g. text analytics)
  - Modules and functions allow for larger, more complex projects
- Jaql seamlessly combines paradigms
  - SQL syntax for working with structured data (à la Hive)
  - Jaql syntax for flow-of-control and semi-structured data (à la Pig)
- Contains features missing from other languages
  - Modules
  - First-class functions
  - Low-level tweaking of 'query plan'

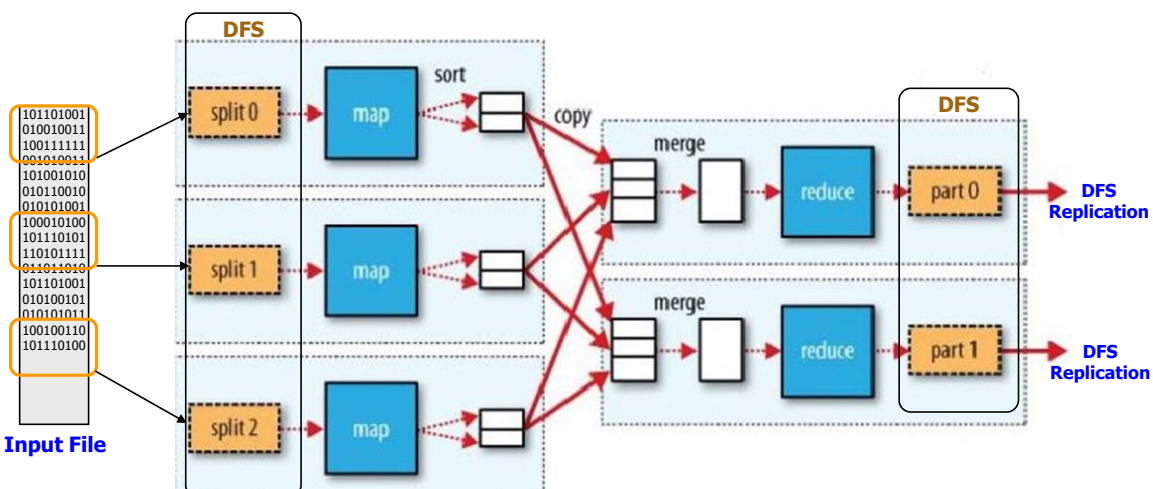
MapReduce overview

- Core MapReduce concepts
  - **Mapping**:
    - Input data (typically file(s)) is read and turned in `<key,value>` pairs
  - **Shuffle**:
    - Data is sorted by `key` and all data sharing the same key is grouped into `<key, (value1, value2, ...)>`
  - **Reduce**:
    - Output from the shuffle is processed, and final resulting `<key, value>` pairs are produced

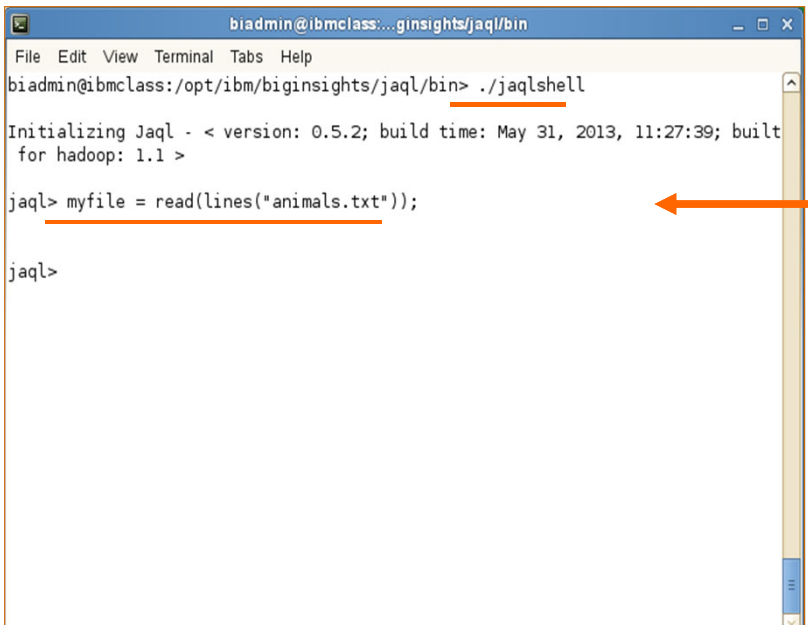


MapReduce and Hadoop

- Process is parallelized by taking advantage of a Distributed File System
  - Input is broken in multiple pieces (splits), typically by block on the DFS
  - Mapping logic is scheduled to run *local to its split*, if possible
  - Shuffle is handled transparently by Hadoop
  - Each reducer works on a different `<key, (value1, value2,...)>` set



Starting up the Jaql Server & Entering Jaql in command line mode

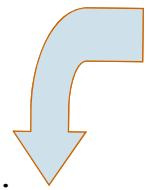


Start JAQL with **jaqlshell** command

Enter JAQL statement, and, when complete statement has been entered,add semi-colon and press enter

Follow up with other JAQL statements, each with a semi-colon

Jaql and MapReduce



cow
dog
bee
cow

```
myfile = read(lines("animals.txt"));
```

```
myfile
  -> group by animal = $
          into { animal, total: count($) }
  -> write(del("animal_count.csv"));
```

bee, 1
cow, 2
dog, 1

Let's do this step by step


```
Jaql> myfile = read(lines("animals.txt"));
```

cow
dog
bee
cow

```
jaql> myfile;
[
  "cow",
  "dog",
  "bee",
  "cow"
]
```

By using the variable that was created (**myfile**), you can see what Jaql has read, *or* would read, whenever the read is instantiated

Referencing a variable by itself will display the variable's contents

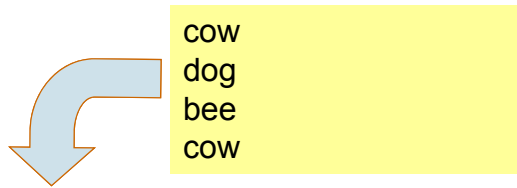
Next step — applying GROUP BY — intermediate results


```
jaql> myfile -> group by animal = $
                  into { animal, total: count($) };
```

```
[
  {
    "animal": "bee",
    "total": 1
  },
  {
    "animal": "cow",
    "total": 2
  },
  {
    "animal": "dog",
    "total": 1
  }
]
```

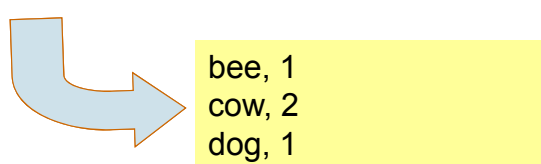
Intermediate results are in JSON format

Final result from Jaql invoking the MapReduce process




```
myfile = read(lines("animals.txt"));

myfile
  -> group by animal = $
          into { animal, total: count($) }
  -> write(del("animal_count.csv"));
```



Jaql and MapReduce — the Rewrite Engine and Explain

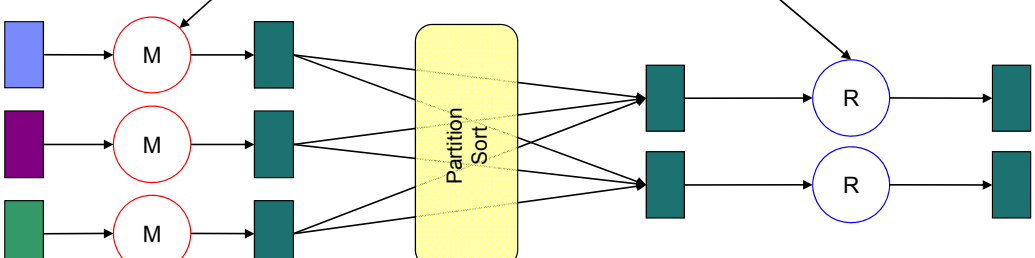
Rewrite Engine

// **Query:** Count # animals in a text file

```
read(lines("animals.txt")
)
  -> group by animal = $
          into { animal,
total: count($) }
  ->
write(del("animal_count.c
sv"));
```

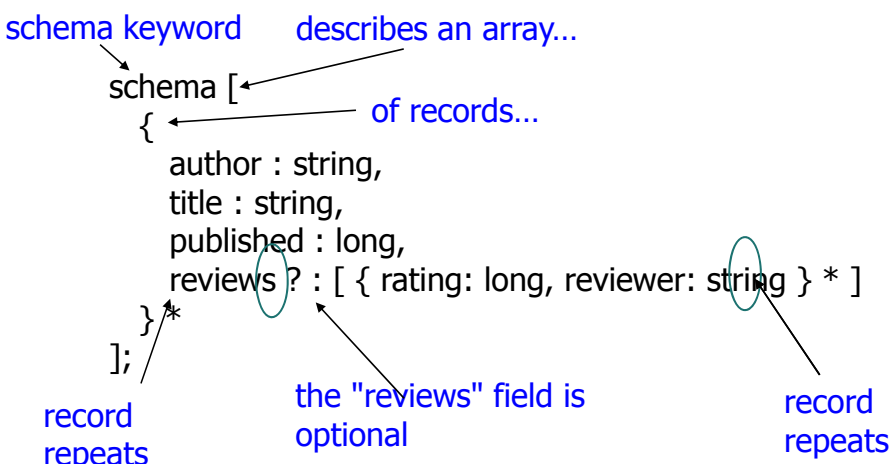
// **Query:** equivalent map-reduce job in Jaql

```
mrAggregate({
 input:  { location: "animals.txt", type: "hdfs" },
 output: { location: "animal_count.csv", type: "hdfs" },
 map:    fn($vals)
           $vals -> transform [$, $],
 agg:    fn($toagg)
           [$toagg -> system::count()],
 reduce: fn($vals)
           [ { animal: animal, total: $vals[0] } ]
```



Jaql schema

- Schema can be used to fully or partially describe/restrict data
  - Required to describe values from data sources that cannot describe themselves
    - (e.g. CSV files)
  - With binary files, schemas optimize storage
  - Describe parameters and return values from functions
  - Any object can be asked its schema with the schemaof() function



Data types

- Types
  - Jaql is a loosely typed language
  - Type is usually inferred by how a value is provided
  - Schemas can be used to force type conversions
  - Many types have a function of the same name to force conversions of a value or variable (e.g. `string()`, `double()`, etc.)

Standard JSON Types

null – null
boolean – true, false
string – "hi"
long – 10
double – 10.2, 10d, 10e-2
array – [1, 2, 3]
record – { a : 1, b : 2 }

Jaql Extensions

decfloat – 3.5m, 3m, 10e-2m
binary – hex("F0A0DEE")
date – date("2001-07-04T12:00:40.002Z")
schema – schema [long *]
function – fn (a, b) a + b
comparator - cmp(x) [x.someField asc]
regex – regex("[^[a-c].*]")

JSON applications outside of Jaql may not understand

Jaql basics

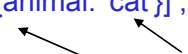
- All Jaql statements are terminated with a semicolon

- Single and double quotes are treated as the same
- Commenting
  - // indicates a comment through the end of the line
  - /* … */ indicate a block of comments
- Supports atomic type variables

```
jaql> num = 10;
jaql> str = 'abc';          /* or could be str = "abc";  */
jaql> flg = true;
```

- Supports complex type variables

```
jaql> array = [1, 2, 'abc', {animal: 'cat'}] ;
```



Mixed types in the same array

- **Referencing a variable by itself will display the variable's contents**

Arrays

- Arrays can be accessed with the [ ] operator (zero based indexed)

```
jaql> a = [1,2,3,4];
jaql> a[1];
2

jaql> a[1:3];
[2,3,4]
```

- Remember, variables are immutable
  - To change an array value, you *must* produce a *new* array

```
jaql > a = replaceElement(a, 1, 10);
jaql> a[1];
10
```

Arrays continued

Most array operations are done via functions

The following is just a subset

Length	count()	**jaql>** `count(['a', 'b', 'c']);` `3`
Get element	index()	**jaql>** `index(['a', 'b', 'c'], 1);` `"b"`
Replace element	replaceElement()	**jaql>** `replaceElement(['a', 'b', 'c'], 1,'z');` `[ "a", "z", "c" ]`
Subset	slice()	**jaql>** `slice(['a','b','c','d'], 1, 2);` `[ "b", "c" ]`
Reverse	reverse()	**jaql>** `reverse(['a', 'b', 'c'];` `[ "c", "b", "a" ]`
Generate	range()	**jaql>** `range(2, 5);` `[ 2, 3, 4, 5 ]`

Records

- Fields within records are accessed using the "." operator

```
jaql> a = {name: "Mary", age: 38, children: ["Brittany", "Jacob"]};
jaql> a.age;
  38
jaql> a.children[0];
  "Brittany"
```

- Fields can be quoted

```
jaql> r={"Title of Book": "The Lincoln Lawyer", author: "Connelly"};
jaql> r."Title of Book";
 "The Lincoln Lawyer"
```

- Variables can be used to access fields

```
 jaql> b="author";
 jaql> r.(b);
  "Connelly"
```

Records – subsetting and projection

- Record subsetting or addition

```
jaql> r = {a: 1, b: 2, c: 3};
jaql> r{.a , .b};
 {a:1, b:2}
jaql> {r.*};
 {a:1, b:2, c:3}
jaql> {r.*, d:4};
 {a:1, b:2, c:3, d:4}
jaql> r{* -.b};
 {a:1, c.3}
jaql> {r{* -.b}, d:4};
 {a:1, c:3, d:4}
```

Subset

Field addition

Field subtraction

Mix-n-match

- Records can be projected

```
jaql> ar = [{a: 1, b: 2}, {a:11,
  b:12}];
jaql> ar[*].a;
 [1,11]
jaql> ar.a;
 [1,11]
jaql> ar[*]{.a };
 {a:1, a:11}
```

[*] is the preferred syntax, as it is more explicit

Record functions

Many record operations are done via functions

The following is just a subset

To array	fields()	**jaql>** `fields({a: 1, b: 2});` `[ ["a",1], ["b",2] ]`
From array	record()	**jaql>** `record([ ["a",1], ["b",2] ]);` `{ a: 1, b: 2 }`
Field names	names()	**jaql>** `fields({a: 1, b: 2});` `[ "a", "b" ]`
Values	values()	**jaql>** `values({a: 1, b: 2});` `[ 1, 2 ]`

Operators

- Basic operators

Arithmetic

```
+ : a + b, "a" + "b"
- : a - b, -a
/ : a / b
* : a * b
```

Only work on scalar types

Boolean

```
and  : a and b
or   : a or b
not  : a and not b,
       not (a and b)
```

Only work on boolean type

Comparison

```
== : a == b
!= : a != b
<  : a < b    <= : a <= b
>  : a > b    >= : a >= b
in : a in ["b", "c", "d"]
isnull : isnull a
```

- Assignments
  - = Normal assignment (a = 10)
  - := Materialized assignment (a := 10)

Lazy/late evaluation

- Return value from most Jaql functions are not materialized or generated until they are needed!
  - The range(N) function returns an array of the numbers from 0 to N-1
  - So why doesn't the following cause us to run out of memory?

```
jaql> x = range(9223372036854775807);
```

- The results from range() are only produced as they are accessed

```
jaql> x = range(9223372036854775807);
jaql> x -> top 2;
[
  0,
  1
]
```

Fetches first 2 entries from an array.We'll cover "->" and "top" soon.

- The := operator forces materialization

```
jaql> x := range(9223372036854775807);
java.lang.OutOfMemoryException
…
```

```
jaql> range(5);
[
  0,
  1,
  2,
  3,
  4
]

jaql>
range(2,5);
[
  2,
  3,
  4,
  5
]
```

Why materialized assignment (:=) ?

- Even file reads are lazy (they just return a descriptor), so without care, many nodes can concurrently contend for a single file:

```
jaql> config_file = read(lines("config.txt"));
jaql> big_mapreduce_function(config_file);
```

Read doesn't actually happen here!

Each mapper/reducer that tries to access config_file will cause a read

- Using := forces the read to happen locally
  - Contents of the read will be passed to the MR job:

```
jaql> config_file := read(lines("config.txt"));
jaql> big_mapreduce_function(config_file);
```

- Note that := can only occur at the top level and not inside of functions

The -> operator

- The -> operator "streams" an array through a function or core operator
  - This is basically designed to enhance readability
- An example
  - The *batch* operator splits an array into an array of arrays
    - The first argument for the *batch* operator is an array
    - The second argument is the size

      batch(*array, size*)

  - Using the -> operator, you can invoke the *batch* operator as follows:

    array -> batch(size);

  - The -> operator implicitly passes the array on the left as the first argument to the function
- The following are equivalent

```
batch(range(10), 5);
range(10) -> batch(5);
```

Expressions

- if

  if (*expr*) *expr1* else *expr2*

```
jaql> y = 10;
jaql> if (y < 20) 3 else 4;
  3
```

- for

  for (*val* in *array*) [*expr*]

```
jaql> ar = [1,2,3,4];
jaql> for (x in ar) [x * 2];
   [2,4,6,8]
```

- Block

  (expr, expr, …)

  - Defines a new variable scope

```
jaql> a = 10
jaql> (a = 1, b=a + 5, b * 2);
  12
jaql> a;
  10
```

Functions

- User-defined functions can be defined in
  - Jaql
  - Java
- Basic definition

```
var = fn (arg1 [, arg2, arg3, …]) expr;
```

- Example

```
jaql> add = fn(x, y) x + y;
jaql> add(1, 2);
  3
```

- Functions are first class objects
  - They can be assigned to variables
  - They can be passed as parameters

```
jaql> apply_fn = fn(op, array) for (val in array) [ op(val)];
jaql> apply_fn( fn(x) x * x, [1, 2, 3]);
  [1, 4, 9]
```

Function parameters

- Can define default parameter values

```
jaql> div = fn(x, y, round=false);
```

  - If a value is not passed for the *round* parameter, then the *round* parameter will assume a value of *false*

- Can pass parameters by name
  - Then the order of parameters is not significant

```
jaql div = fn(x, y, round=false);
jaql div(y = 1.2, x = 5);
jaql div(5, 1.2, true);
```

Functions – additional

- Schema syntax is used to
  - Restrict input types
  - Define return types

```
jaql> my_funct = fn(rec : {a : long}, val : long) :{a : long, x : long} … ;
```

- Recursive functions
  - Direct recursion is not allowed
  - A trick to get by this restriction
    - Pass the function to itself

- Jaql allows functions to be implemented in Java
  - User-defined function (UDF) can be called like a regular Jaql function
  - User- defined aggregate (UDA) can be used to implement an aggregation function for data grouping

Why Jaql core operators

- MapReduce coding can become complex
  - Joining data from multiple files
  - Filtering data
  - Unioning data
- MapReduce programming requires a particular skill set
- Without the core operators
  - Programmers would be recreating the work done by other programmers
- Core operators act as a library
  - The join operator was coded once
    - Is then shared across installations
- Core operators allows non-programmers to more easily analyze big data

Core operators

- Core operators manipulate streams (arrays) of data, much in the way SQL clauses interact with data
  - These operators require the use of the -> operator

- Jaql core operators
  - expand
    - Flattens nested arrays
  - filter
    - Similar to a WHERE clause, filtering array data
  - group
    - Groups one or more arrays one based upon key values, applying an aggregate expression
  - sort
    - Sorts the contents of an array
  - top
    - Returns the first *N* elements of an array
  - transform
    - Transforms elements of the input array to produce a modified output array

Core operators – common behavior

- All core operators allow you to apply logic to each entry in an array ("row")
  - The special variable `$` represents the "current" array value being evaluated

```
jaql> data = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ];
jaql> data -> filter 3 <= $ <= 6;
[ 3, 4, 5, 6 ]
```

  - Alternatively, the `each` clause can be used to provide a name different than `$`

```
jaql> data = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ];
jaql> data -> filter each num (3 <= num <= 6);
[ 3, 4, 5, 6 ]
```

Core operators - expand

- The expand operator flattens nested arrays

```
jaql> data = [ [ 1, 2, 3], [4, 5, 6], [7, 8, 9] ];
jaql> data -> expand;
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

- In addition you can apply an expression to each nested array
  - Results from expression are appended to form the new array

```
jaql> data -> expand (slice($, 0, 0));
[ 1, 4, 7 ]
```

- The `each` clause can be used to specify a name other than `$`

```
jaql> data -> expand each subarray (slice(subarray, 0, 0));
[ 1, 4, 7 ]
```

Core operators – filter

- Filter allows you to selectively filter out array entries

```
jaql> data = [ {fname:"scott", age:42}, {fname:"sally", age:21},
               {fname:"sam",age:12} ];

jaql> data -> filter $.age < 20 or $.name == "sally";
[{fname:"sally", age:21},{fname:"sam",age:12} ];

jaql> data -> filter each person (person.fname in ["scott","sam"]);
[{fname:"scott", age:42},{fname:"sam",age:12} ];
```

- Filter supports the following predicates

```
and    : a and b
or     : a or b
not    : a and not b,
         not (a and b)
```

```
==  : a == b
!=  : a != b
<   : a < b     <=  : a <= b
>   : a > b     >=  : a >= b
in  : a in ["b", "c", "d"]
```

Core operators – transform

- The transform operator allows you to manipulate the values in an array ("project" in DBMS vernacular)
  - An expression is applied to each element in the array
  - The result of the expression is the next element in the output array

```
jaql> recs = [ {a: 1, b: 4},  {a: 2, b: 5},  {a: -1, b: 4} ];
jaql> recs -> transform $.a + $.b;
[ 5, 7, 3 ]

jaql> recs -> transform { sum: $.a + $.b };
[ { sum: 5 }, { sum: 7 }, { sum: 3 } ]
```

- As usual `each` can be applied if you don't like `$`

```
jaql> recs -> transform each rec { sum: rec.a + rec.b };
[ { sum: 5 }, { sum: 7 }, { sum: 3 } ]
```

Core operators – group

- Group comes in two distinct forms
  - Single array group
    - Group and aggregate records contained in a single array
  - Co-group
    - Effectively performs a join between 2 or more arrays using a common grouping key

- There are a number of built-in aggregation functions beyond the basic ones (`min`, `max`, `avg`)
  - See the BigInsights Information Center

- Externally, there is nothing special about aggregates
  - They are functions that work over arrays and return a value:

```
jaql> min([1, 2, 3, 4, 5]);
1
```

Core operators – group format (single)

- Syntax

```
>>-A-- ->--group--+--------------------+------------------------>
                  '-each--<elementVar>-'
>--+-------------------------------------------------------+-------->
   '-by--<groupKeyVar>--=--<groupExpr>-+----------------+
                                       '-as--<groupVar>-'
>--into--<aggrExpr>--;-----------------------------------------><
```

- Where:
  - **A**
    - The array over which you are grouping
  - **elementVar**
    - Allows you to specify a name other than `$` to use when referring to elements of **A** within **groupExpr**
  - **groupKeyVar**
    - A name by which your grouping expression can be referred
  - **groupExpr**
    - An expression that produces the key by which you will be grouping
  - **groupVar**
    - The name you wish to refer to the array containing the records that were grouped by your grouping expression
      - If not provided then `$` refers to this array
  - **aggrExpr**
    - Your aggregate expression. Called once for each group, it can use **groupKeyVar** and **groupVar** to produce a value

Note the subtle distinction!

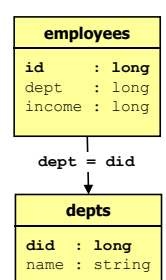
Core operators – group sample data

- In our examples, we will work with the following sets of data:
  - Employee information

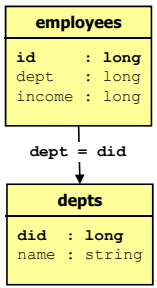
```
employees = [
  {id:1, dept: 1, income:12000},  {id:2, dept: 1,
income:13000},
  {id:3, dept: 2, income:15000},  {id:4, dept: 1,
income:10000},
  {id:5, dept: 3, income:8000},   {id:6, dept: 2, income:5000},
  {id:7, dept: 1, income:24000}
];
```

  - Department information

```
depts = [
  {did: 1, name: "development"},
  {did: 2, name: "marketing"},
  {did: 3, name: "sales"}
];
```



Core operators – group (single) (1 of 3)



- Computing an aggregate over a whole set

```
jaql> employees -> group into count($);
[ 7 ]
```

This `$` refers to the current element in `employees`

- Calculate total salaries by department

```
jaql> employees
        -> group by dept_id = $.dept
             into { dept_id, total_income : sum($[*].income) };

[ { dept_id: 2, total_income: 20000 },
  { dept_id: 1, total_income: 59000 },
  { dept_id: 3, total_income: 8000 }
]
```

This `$` refers to the array of records matching the current key

Core operators – understanding grouping

- The grouping expression:

```
group by dept_id = $.dept
```

- Internally produces an array of { key, [value,value,…] } for each element of the input array that has the same key. So we get:

```
[
  { dept_id: 1, $: [ {id:1, dept: 1, income:12000},
                     {id:7, dept: 1, income:24000},
                     {id:2, dept: 1, income:13000} ] },
  { dept_id: 2, $: [ {id:3, dept: 2, income:15000},
                     {id:6, dept: 2, income:5000} ] },
  { dept_id: 3, $: [ {id:5, dept: 3, income:8000} ] }
]
```

Note! There is *nothing* special about the `$`, it is a legal character for variables.

- Then, the aggregate expression:

```
into { dept_id, total_income : sum($[*].income) };
```

There's that syntax for extracting a field from an array of records again!

- Is called once for each unique key, almost as if you had done:

```
-> transform { dept_id, total_income : sum($[*].income) };
```

Core operators – group (single) (2 of 3)

Here's the same query with more explicit syntax

```
jaql> employees
        -> group each emp by dept_id = emp.dept as groupedEmps
             into { dept_id, total_income : sum(groupedEmps[*].income)};
```

```
[ { dept_id: 2, total_income: 20000 },
  { dept_id: 1, total_income: 59000 },
  { dept_id: 3, total_income: 8000 }
]
```

- Where
  - **emp**
    - The name by which each employee record will be referred in the grouping expression
  - **dept_id**
    - Is the name by which we will refer to the key (group) that is produced by the grouping expression
  - **groupedEmps**
    - Is the name by which the array of grouped employeed records will be referred in the aggregate expression

Core operators – group (single) (3 of 3)

- Note that the key need not be a simple value

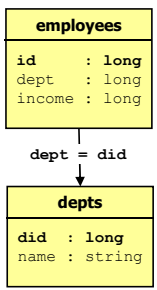
```
jaql> sales
      -> group by key = { $.state, $.product }
           into { key.state, key.product, sales: sum($[*].qty) +
sum($[*].cost) };
```

Slightly simplified

```
jaql> sales
      -> group by key = { $.state, $.product }
           into { key.*, sales: sum($[*].qty) + sum($[*].cost)
};
```

Core operators – co-groups (1 of 3)

- The group operator be applied to more than one input array
  - Each array is provided a grouping expression
  - All grouping expressions must output the same result
  - For each unique group (across all arrays) the aggregate expression receives an array of records for each array in the input expressions



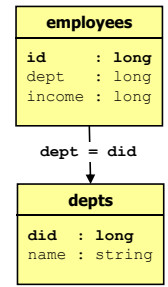
- Joining employees and departments

All groups must have the same name

And the same basic structure (schema)

```
jaql> group employees by g = $.dept as es,
                depts       by g = $.did  as ds
      into { dept:      g,
             deptName: ds[0].name,
             emps:      es[*].id,
             numEmps: count(es) };
```

Core operators – co-groups (2 of 3)



- Our grouping expression:

```
group employees by g = $.dept as es,
       depts     by g = $.did  as ds
```

  - Creates something resembling the following:

Group

Records from employees that fell into the group

Records from depts that fell into the group

```
[
  { g:  1,
    es: [ {id:1, dept: 1, income:12000}, {id:7, dept: 1,
income:24000},
          {id:2, dept: 1, income:13000} ],
    ds: [ {did: 1, name: "development"} ] },
  { g:  2,
    es: [ {id:3, dept: 2, income:15000}, {id:6, dept: 2,
income:5000} ],
    ds: [ {did: 2, name: "marketing"} ] },
  { g:  3,
    es: [ {id:5, dept: 3, income:8000} ],
    ds: [ {did: 3, name: "sales"} ] }
]
```

- For each of the above, the aggregate expression is executed:

```
into { dept: g,  deptName: ds[0].name,  emps:  es[*].id,  numEmps: count(es) };
```

Core operators – co-groups (3 of 3)

- Taking one record as an example:

```
{ g:    1,
  es:  [ {id:1, dept: 1, income:12000}, {id:7, dept: 1, income:24000},
         {id:2, dept: 1, income:13000} ],
  ds:  [ {did: 1, name: "development"} ] },
```

- And our aggregate expression:

```
into { dept: g,  deptName: ds[0].name,  emps:  es[*].id,  numEmps: count(es) };
```

- Results in

```
{  dept: 1,
   deptName: "development",
   emps:  [1, 7, 2],
   numEmps: 3 }
```

Core operators – join

- The `join` operator joins two or more arrays
  - Allows for natural, left-outer, and right-outer joins
- The syntax is:

```
>>-join--+----------+--+------------+--<A1>--,------------------>
         '-preserve-'  '-<var1>--in-'
>--------+----------+--+------------+--<A2>--...----------------->
         '-preserve-'  '-<var2>--in-'
>--------+----------+--+------------+--<An>--------------------->
         '-preserve-'  '-<varn>--in-'
>--where--<joinExpr>--into--<joinOut>--;----------------------><
```

- Where:
  - preserve
    - Indicates that values in the array will appear whether or not another array has a matching value
      - Allows outer join behavior
  - A1-AN
    - The arrays to be joined together
  - var1-varN
    - Establishes an alias to be used to refer to the array name
  - joinExpr
    - The WHERE clause that specifies the join
  - joinOut
    - An expression that produces a value that results from the join

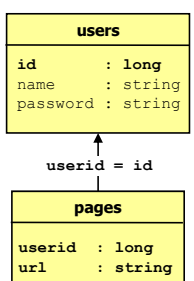
Core operators – join sample data

- Given the following sample data:
  - System users

```
users = [
  {name: "Jon Doe", password: "asdf1234", id: 1},
  {name: "Jane Doe", password: "qwertyui", id: 2},
  {name: "Max Mustermann", password: "q1w2e3r4", id:
3}
];
```

  - Web pages they have accessed

```
pages = [
  {userid: 1, url:"code.google.com/p/jaql/"},
  {userid: 2, url:"www.cnn.com"},
  {userid: 1, url:"java.sun.com/javase/6/docs/api/"}
];
```



Core operators – join sample data

- Returns the pages accessed by each user:

```
jaql> join users, pages where users.id == pages.userid
        into {users.name, pages.*};
```

- Results:

```
[
  { name: "Jane Doe", url: "www.cnn.com", userid: 2  },
  { name: "Jon Doe", url: "code.google.com/p/jaql/", userid: 1 },
  { name: "Jon Doe", url: "java.sun.com/javase/6/docs/api/", userid: 1 }
]
```

- Note that, unlike group, the into expression is called on each match
- The in clause can be used to alias the name of an array

```
jaql> join u in users, p in pages where u.id == p.userid
        into {u.name, p.*};
```

Core operators – join outer joins

- The preserve keyword forces an array to return results, even when no other array matches

```
jaql> join preserve u in users, p in pages where u.id == p.userid
        into {u.name, p.*};
```

- Results:

```
[
  { name: "Jane Doe", url: "www.cnn.com", userid: 2  },
  { name: "Jon Doe", url: "code.google.com/p/jaql/", userid: 1 },
  { name: "Jon Doe", url: "java.sun.com/javase/6/docs/api/", userid: 1 },
  { name: "Max Mustermann" }
]
```

- For more "SQL-Like" results:

```
jaql> join preserve u in users, p in pages where u.id == p.userid
        into {u.name, url: p.url, userid: p.userid};
```

```
    [
      ...
      { name: "Max Mustermann", url: null, userid: null }
    ]
```

Core operators – sort

- The sort operator allows sorting of arrays

```
jaql> people = [ { name: "scott", age: 42 }, { name: "jake", age: 14 },
                 { name: "sam", age: 12 } ];

jaql> people -> sort by [ $.name ];
 [ { name: "jake", age: 14 }, { name: "sam", age: 12 }, { name: "scott", age: 42 } ];
```

- More than one field may be specified for sorting and the asc or desc modified can specify direction of sort

```
jaql> people -> sort by [ $.name desc, $.age asc ];
[{ name: "scott", age: 42 }, { name: "sam", age: 12 }, { name: "jake", age: 14 } ];
```

- Sorting can be performed with any expression:

```
jaql> people -> sort each p by [ substring(p.name, 0, 1), p.age asc ];
[ { name: "jake", age: 14 }, { name: "sam", age: 12 }, { name: "scott", age: 42 } ];
```

Core operators – top

- The top operator returns the first *n* rows of its input array

```
jaql> data = [1, 2, 3, 4, 5, 6, 7, 8, 9];
jaql> data -> top 2;
[ 1, 2 ]
```

- It can also combine the functionality of `sort` with the `by` clause

```
jaql> data -> top 3 by [ $ desc ];
[ 9, 8, 7 ]
```

Jaql SQL

- SQL statements may be included as part of the Jaql flow
- Much of SQL '92 SELECT syntax is supported
- "Tables" are Jaql data sources (arrays of records)

```
read(hdfs('books'))
  -> filter $.year >= 1995
  -> group by pub = $.publisher
     into { name : pub,
            num : count($) }
  -> write(hdfs('summary'));
```

```
books = read(hdfs('books'))
   -> transform check ($, pub_schema)
(SELECT b.publisher "name", count(*) "num"
FROM books b
WHERE b.year >= 1995
GROUP BY b.publisher)
  -> write(hdfs("summary"));
```

- Compiles into Jaql

Jaql SQL - Examples

- Sample data

```
jaql> T1 = [ { a : 1 }, { a : 2 } ];
jaql> T2 = [ { b : 1 }, { b : 5 } ];
```

- Joins are supported

```
jaql> select * from T1, T2 where t1.a = t2.b;
```

- Fully connected joins plus local predicates are supported

```
jaql> select * from T1, T2 where t1.a = t2.b and t2.b > 2;
```

- Cross products are supported

```
jaql> select * from T1,T2;
```

- Joins may only be performed using equality predicates

```
jaql> select * from T1, T2 where t1.a < t2.b;
jaql> select * from T1, T2 where t1.a >= t2.b;
```

ERROR!

- Correlated subquery to work around

```
jaql> select * from T1, (select * from T2 where t1.a < t2.b) V;
```

Jaql SQL – case-sensitivity

- Once referenced in a FROM clause a Jaql object is not case-sensitive, otherwise it is case-sensitive
- Given the “table” T, defined as:

```
jaql> T = [ { a : 1 }, { a : 2 }, { a : 3 } ];
```

- Non case-sensitive reference to row qualifier works:

```
jaql> select t.a from T;
```

- Non case-sensitive reference to column/field name works:

```
jaql> select t.A from T;
```

- Because T is a Jaql variable, it is case-sensitive

```
jaql> select t.a from t;
parse error: java.lang.IndexOutOfBoundsException: variable is not defined: t
```

- Same for Jaql variables not in the from clause

```
jaql> v = 20;
jaql> select V, t.a from T;
parse error: java.lang.IndexOutOfBoundsException: variable is not defined: V
```

Jaql SQL – functions and Jaql expressions

- Any Jaql functions can be utilized within SQL

```
jaql> square = fn (x) x * x;
jaql> data = [ { a : 1 }, { a : 2 }, { a : 3 } ];
jaql> select square(a) "a" from data;
[ { a : 1 }, { a : 4 }, { a : 9 } ]
```

- Jaql expressions may be inserted into SQL using `jaql()`

```
jaql> apply = fn (op, val) op(val);
jaql> select apply(jaql(fn (x) x * x), a) "a" from data;
[ { a : 1 }, { a : 4 }, { a : 9 } ]
```

Jaql SQL - Schemas

- SQL statements *require* a schema at compile time
- Querying results from input adapters requires a schema to be provided (even if the adapter doesn't need it!)
  - From `del()` the schema is automatically picked up

```
jaql> data = read(del("test.csv", schema = schema {name:string, age:long}));
jaql> select name, age from data;
```

  - Some input adapters do not properly present their schema as they should (this is a bug):

```
jaql> data = read(seq("test.seq", schema = schema {name:string, age:long}));
jaql> select name, age from data;
sql requires only records instead of :schema nonnull
```

  - This can be remedied with `check()`

```
jaql> data = read(seq("test.seq"))
          -> transform check($, schema {name:string, age:long});
jaql> select name, age from data;
```

Jaql and MapReduce basics

- The read(), along with several other Jaql functions, will attempt to parallelize its operation via MapReduce, *if…*
  - The source of data (adapter) is splittable
  - The operations performed upon the results of the operation do not prevent MapReduce from being deployed
- Streaming data sources, such as JDBC connections are not splittable
  - However, there is a DB2 partitioning module that allows splitting over DB2 partitions!
- Certain functions involved in the operation can prevent splitting
  - E.g. prevElement() – Allows you to "window" over an array, working with the current and previously visited value
- Certain file formats, such as some compression formats are not splittable
  - e.g. .gz or .bz2
- The explain command will tell you how Jaql is approaching your query…

Jaql and MapReduce - explain

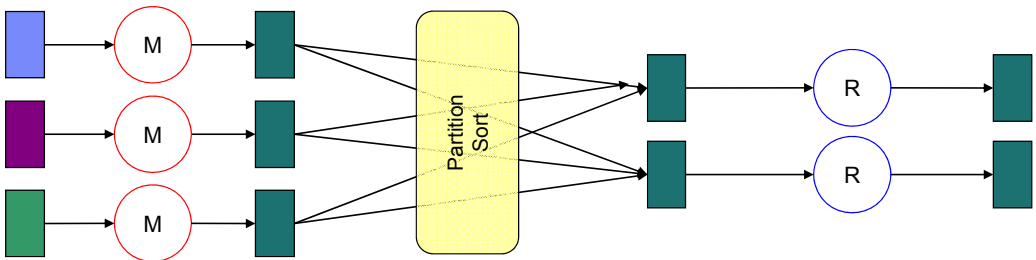
- Most Jaql statements compile into lower-level Jaql statements
- These statements can be invaluable in understanding how Jaql is solving a problem

```
explain read(hdfs('books'))
  -> filter $.year >= 1995
  -> group by pub = $.publisher
     into { name : pub,
            num : count($) }
  -> write(hdfs('summary'));
```

```
system::mrAggregate(
{
  "input" : { "type": "hdfs",
              "location": "books" }
  "output": { "type": "hdfs",
              "location": "summary" },
  "map"    : fn($recs) (
     $recs
       -> filter $."year" >= 1995
       -> transform [$."publisher", $]),
  "aggregate" :
     fn(pub, $toagg) [ $toagg -> system::count() ],
  "final" :
     fn(pub, $vals) (
       [ { "publisher" : pub,
           "num" : $vals -> system::index(0) } ]),
});
```

- Key re-writes to look for:
  - mapReduce() – Logic is turned into a MapReduce job
  - mrAggregate() – Logic is turned into a Map+Combiner+Reduce job
  - None of above – Jaql determined it cannot use MR on your query

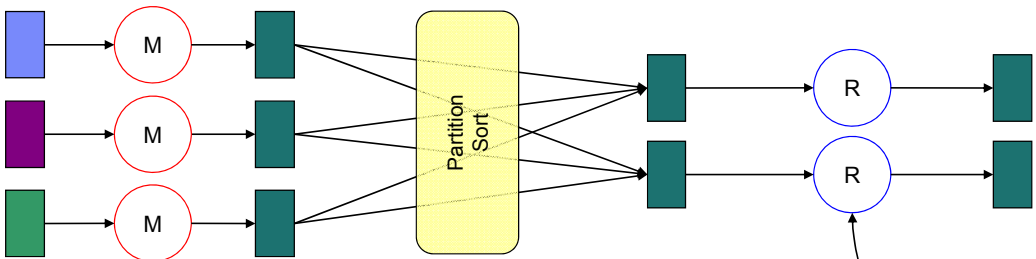
Jaql and MapReduce – map




```
explain read(hdfs('books'))
  -> filter $.year >= 1995
  -> group by pub = $.publisher
     into { name : pub,
            num : count($) }
  -> write(hdfs('summary'));
```

```
system::mrAggregate(
{
  "input" : { "type": "hdfs",
              "location": "books" }
  "output": { "type": "hdfs",
              "location": "summary" },
  "map"    : fn($recs) (
     $recs
       -> filter $."year" >= 1995
       -> transform [$."publisher", $]),
  "aggregate" :
     fn(pub, $toagg) [ $toagg -> system::count() ],
  "final" :
     fn(pub, $vals) (
       [ { "publisher" : pub,
           "num" : $vals -> system::index(0) } ]),
});
```

Jaql and MapReduce - reduce




```
system::mrAggregate(
{
  "input" : { "type": "hdfs",
              "location": "books" }
  "output": { "type": "hdfs",
              "location": "summary" },
  "map"   : fn($recs) (
     $recs
       -> filter $."year" >= 1995
       -> transform [$."publisher", $]),
  "aggregate" :
     fn(pub, $toagg) [ $toagg -> system::count() ],
  "final" :
     fn(pub, $vals) (
       [ { "publisher" : pub,
           "num" : $vals -> system::index(0) } ]),
 });
```

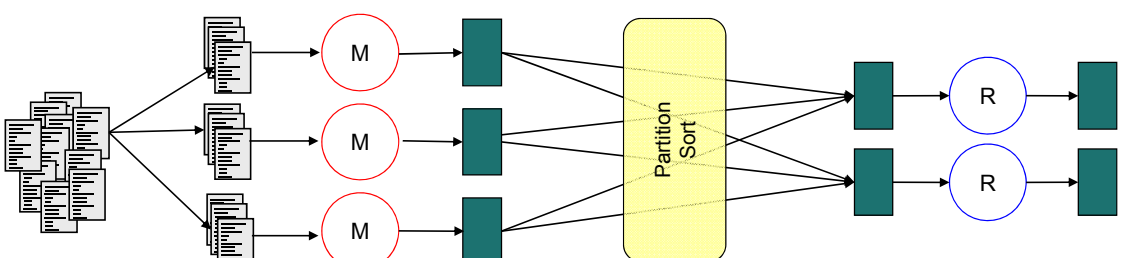
```
explain read(hdfs('books'))
  -> filter $.year >= 1995
  -> group by pub = $.publisher
     into { name : pub,
            num : count($) }
  -> write(hdfs('summary'));
```

Jaql and MapReduce – explicit parallelism

- The `arrayRead()` function explicitly sends each element of an array to a mapper
- Frequently used with `batch()` to create explicit task lists

```
jaql> ls ("/path/to/files/*.csv").path
        -> batch (10)
        -> arrayRead()
           -> read(del(location = $, schema =
                  schema { fname: string, age: long } ))
           -> group g = …
```

1. List all *.csv files in a directory
2. Create an array of arrays of 10 files each
3. Send each array to a mapper to work on
4. Each mapper reads and aggregates



MapReduce – job configuration

- Jaql MR jobs default to configuration in `$HADOOP_CONF_DIR`
- Configuration options can be explicitly set/overridden with `setOptions()`

```
jaql> setOptions( { conf: {
        "mapred.job.name": "Jaql Job",
        "mapred.map.tasks": "1",
        "mapred.reduce.tasks": "5" } });
```

- `loadJobConf()` can be used to read a configuration file

```
jaql> setOptions(loadJobConf("file:///path/to/mapred-site.xml"));
```

- `getOptions()` displays options that have been explicitly set

```
jaql> getOptions();
{
  "conf": {
    "mapred.job.name": "Jaql Job",
    "mapred.map.tasks": "1",
    "mapred.reduce.tasks": "5"
  }
}
```

Jaql and MapReduce – native MR jobs (1 of 2)

- Jaql can launch native (Java) MR jobs with `nativeMR()`

```
nativeMR( {
        "mapred.job.name": "DoSomeBigThing",
        "mapred.input.dir": "/path/to/*.txt",
        "mapred.input.format.class": "org.apache.hadoop.mapred.TextInputFormat",
        "mapred.output.format.class": "org.apache.hadoop.mapred.TextOutputFormat",
        "mapred.map.mapper.class": "com.foo.DoSomeBigThingMapper",
        "mapred.output.dir" : "/path/to/output/dir",
        "mapred.output.key.class" : "org.apache.hadoop.io.IntWritable",
        "mapred.output.value.class" : "org.apache.hadoop.io.Text",
        "mapred.reduce.tasks" : "0"
    },
    { useSessionJar: true, apiVersion: "0.0" });
```

- Where:
  - **useSessionJar**
    - If true all class/jars used by Jaql are made available to the job
  - **apiVersion**
    - If 0.0, then the old Hadoop job APIs are utilized, otherwise the new APIs are used

Jaql and MapReduce – native MR jobs (2 of 2)

- Jars required for native MR jobs can be made available with:
  - Calling `addClassPath()`

```
jaql> addClassPath("lib/foo.jar");
jaql> addClassPath("lib/bar.jar");
jaql> nativeMR( { … }, { useSessionJar: true, apiVersion: "0.0" });
```

  - The `--jars` flag to JaqlShell

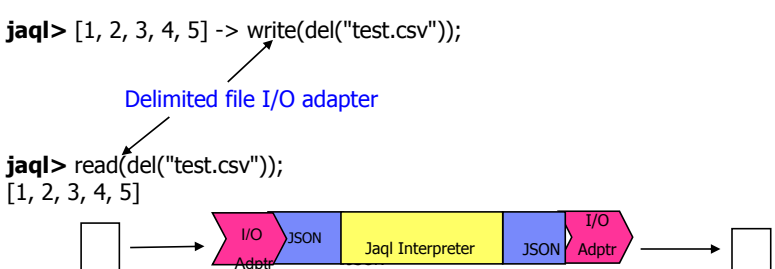
Note the "," between jars

```
$ jaqlshell --jars lib/foo.jar,lib/bar.jar
jaql> nativeMR( { … }, { useSessionJar: true, apiVersion: "0.0" });
```

- Jars are distributed to the mappers via the Hadoop distributed cache
  - For large clusters this can take a long time
  - Try to keep jars to the minimal required set

Jaql I/O

- I/O in Jaql is performed through I/O adapters
  - Adapters return a description of how to access and process a data source
  - Handles accessing storage system (e.g. HDFS, GPFS, JDBC, etc.)
  - Handles processing of data in it's stored format
  - Responsible for translating to/from arrays of JSON records
- Once created the I/O adapter is passed into an I/O function (e.g. read() or write()) to perform the actual operation



Jaql I/O adapter operations

- Jaql provides a handful of operations that can be performed on I/O adapters:
  - `read()`
    - For adapters that are partitionable (can be split), a MapReduce job may be utilized to read the file
  - `localRead()`
    - Forces the I/O processing to take place in the local Jaql instance
    - Useful for testing or working with files in which MapReduce would be overkill
  - `write()`
    - Writes data to the I/O adapter
- Jaql provides other mechanisms to perform explicit MapReduce partitioning

Jaql I/O – I/O adapters

- Unless noted otherwise, most file I/O adapters use Hadoop I/O
  - They default to the default filesystem configured for your Hadoop/BI environment (`fs.default.name`)
  - The current working directory from your shell is ignored
  - This can lead to unanticipated behavior:

```
shell $ cat my_data.csv
1, scott
2, bert
3, tina

shell $ jaqlshell
jaql> localRead(del("my_data.csv"));
...
Input path does not exist: hdfs:/user/idcuser/my_data.csv
```

HDFS "home" directory for current user

Jaql I/O – I/O adapters (cont.)

- Many file I/O adapters allow a URI to force the location of a file:

```
jaql> localRead(del("file:///home/idcuser/my_data.csv"));
[ [1, "scott"], [2, "bert"], [3, "tina" ] ]
```

- Most file adapters will also recognize common filename extensions:

```
jaql> localRead(del("file:///home/idcuser/my_data.csv.gz"));
[ [1, "scott"], [2, "bert"], [3, "tina" ] ]
```

File is compressed with GNU zip

  - This includes write operations as well
  - Common recognized extensions

```
.zip    .bz2
.gz     .deflate
.cmx
```

- Many can process directories of files as well:

```
jaql> localRead(del("file:///home/idcuser/data/*.csv"));
[ [1, "scott"], [2, "bert"], [3, "tina" ], [4, "fred"], [5, "ted"] ]
```

file1.csv file2.csv

Jaql I/O – I/O adapter arguments

- Most I/O adapters have the same basic definition:

```
fn name (
  location : string,
  schema : schema,
  inoptions : { * },
  outoptions : { * },
  options : { * }
)
```

- Where:
  - location
    - Path to the file(s)
  - schema
    - A Jaql schema describing the records to be produced by the file
  - inoptions
    - A record describing options/configurations for reading
    - For most adapters, this is used only to override implementing classes
  - outoptions
    - A record describing options/configurations for writing
    - For most adapters, this is used only to override implementing classes
  - options
    - HELP
- Some adapters have additional arguments of their own

Jaql I/O – I/O adapters arguments (cont.)

- What does an I/O adapter return?
  - A JSON record describing how the file is to be accessed
  - Most fields can be customized to create your own input adapter

```
jaql> del("foo.csv");
{
  "location": "foo.csv",
  "inoptions": {
     "adapter": "com.ibm.jaql.io.hadoop.DelInputAdapter",
     "format": "com.ibm.biginsights.compress.mapred.CompressedTextInputFormat",
     "configurator": "com.ibm.jaql.io.hadoop.FileInputConfigurator",
     "converter": "com.ibm.jaql.io.hadoop.converter.FromDelConverter",
     "delimiter": ",",
     "quoted": true,
     "ddquote": true,
     "escape": true
  },
  "outoptions": {
     "adapter": "com.ibm.jaql.io.hadoop.DelOutputAdapter",
     "format": "org.apache.hadoop.mapred.TextOutputFormat",
     "configurator": "com.ibm.jaql.io.hadoop.FileOutputConfigurator",
     "converter": "com.ibm.jaql.io.hadoop.converter.ToDelConverter",
     "delimiter": ",",
     "quoted": true,
     "ddquote": true,
     "escape": true
  },
  "options": {}
}
```

Instantiated by read() when it comes time to open the file(s)

Properties that configure the classes above

Jaql I/O – schemas

- Self-describing data sources transparently convert to/from JSON

```
jaql> read(jdbc::prepare(conn, query = "SELECT fname, lname, age FROM people"));
[
  { fname : "Fred", lname : "Johnson", age : 20 },
  { fname : "Sally", lname : "Farmsworth", age : 47 }
]
```

- Some formats require a little assistance by providing a Jaql schema

```
jaql> read(del(location="people.csv", schema =
         schema { fname : string, lname : string, age : long }));
[
  { fname : "Fred", lname : "Johnson", age : 20 },
  { fname : "Sally", lname : "Farmsworth", age : 47 }
]
```

- Some operations, such as Jaql SQL, need to know the schema at compile time
  - You may need to provide a schema even for self-describing data sources

Jaql I/O – line files

- The `lines()` adapter allows you to read/write of lines of text
  - Returns an array of lines of text
  - Is partitionable for MapReduce (blocks rounded to nearest EOL)
  - No significant configuration options
    - Custom class for `inoptions = { converter: … }` could be used to parse the input line further

```
shell $ cat nums.txt
1
2
3
shell $ jaqlshell
jaql> localRead(lines(location="file:///path/to/nums.txt", schema = schema long));
[
  1,
  2,
  3
]
```

Note! Schema used to force the lines to be interpreted as numbers.

Jaql I/O – delimited files

- The `del()` adapter allows processing of delimited files
  - Is partitionable for MapReduce (blocks rounded to nearest EOL)
- Additional available arguments
  - delimiter
    - A single character indicating the delimiter (default: ",")
  - quoted
    - Boolean indicating whether or not strings should be quoted (default: true)
  - ddquote
    - Boolean that controls whether or not the escape character is a double quote (true) or a backslash (false) (default: true)
  - escape
    - Boolean that controls whether or not reserved characters (comma or double quote) are escaped (default: true)

```
jaql> [ [ "hi", "there" ], [ "boy", "howdy" ]] -> write(del("test.txt",
delimiter="_"));
jaql> hdfsShell("-cat test.txt");
hi_there
boy_howdy
```

Handy function! This is the equivalent of running:
`hadoop fs -cat test.txt`

Jaql I/O – delimited files (cont.)

- Without a schema, arrays of arrays are returned

```
shell $ cat people.txt
Fred,Johnson,20
Sally,Farmsworth,47
```

```
shell $ jaqlshell
jaql> localRead(del(location="file:///path/to/people.txt"));
[ [ "Fred", "Johnson", "20" ], [ "Sally", "Farmsworth", "47" ] ]
```

- An array schema can force the types in the arrays returned

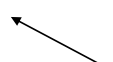
```
jaql> localRead(del(location="file:///path/to/people.txt", schema =
          schema [ string, string, long ] ));
[ [ "Fred", "Johnson", 20 ], [ "Sally", "Farmsworth", 47 ] ]
```

- A record schema can be used to generate JSON records

```
jaql> localRead(del(location="file:///path/to/people.txt", schema =
          schema { fname : string, lname : string, age : long} ));
[ { fname: "Fred", lname: "Johnson", age: 20 },
  { fname: "Sally", lname: "Farmsworth", age: 47 } ]
```

Jaql I/O – binary sequence files

- The `seq()` adapter can be used to work with Hadoop sequence files
  - Binary file format
  - Partitionable to nearest "sync point" – generated every few records
  - Schema information is stored with each record
  - No need to provide schema for read operations in most cases
  - The default format is not generally usable outside of Jaql
  - Format may not be readable between Jaql releases!



Important!!

```
jaql> [ { fname: "Fred", lname: "Johnson", age: 20 },
        { fname: "Sally", lname: "Farmsworth", age: 47 } ]
          -> write(seq(location = "test.seq"));

jaql> localRead(seq(location = "test.seq"));
[ { "Fred", "Johnson", "20" }, { "Sally", "Farmsworth", "47" } ]
```

Jaql I/O – text sequence files

- Sequence files can be made more portable
  - A converter can be used to read or write text JSON records

```
jaql> data = [ { fname: "Fred", lname: "Johnson", age: 20 },
               { fname: "Sally", lname: "Farmsworth", age: 47 } ];

jaql> data -> write(seq(location = "test.seq", outoptions = {
        converter: "com.ibm.jaql.io.hadoop.converter.ToJsonTextConverter",
        configurator: "com.ibm.jaql.io.hadoop.TextFileOutputConfigurator"}));

jaql> localRead(seq(location = "test.seq", inoptions = {
        converter: "com.ibm.jaql.io.hadoop.converter.FromJsonTextConverter"}));
[ [ "Fred", "Johnson", 20 ], [ "Sally", "Farmsworth", 47 ] ]
```

- Other conversion examples can be found in the BigInsights Information Center
- Jaql's Avro module can be used for binary portability

Jaql I/O – compact binary sequence files

- The `jaqltemp()` adapter is just like `seq()` except:
  - A schema is required
  - The schema is written only once for the whole file
  - Significantly more compact than regular `seq()`
  - Like `seq()`, the file format can change between Jaql releases!
  - Thus, should only be used for temporary, short term, storage

Jaql I/O – other adapters

- Most Jaql modules provide additional adapters, such as:
  - Avro
    - Provides a portable mechanism for reading/writing binary records
  - JDBC
    - Read/write from a remote database
  - HBase
    - Work with HBase tables
  - Text analytics
    - Work with unstructured/semistructured text documents
  - Etc.

Checkpoint (1 of 2)

1. JAQL is a JSON query language
   a. True
   b. False

2. JAQL can be used for _____
   a. Data Transformation
   b. Analysis
   c. Generating Reports
   d. Visualizations

Checkpoint solution (1 of 2)

1. JAQL is a JSON query language
   a. True
   b. False

2. JAQL can be used for _____
   a. Data Transformation
   b. Analysis
   c. Generating Reports
   d. Visualizations

Checkpoint (2 of 2)

3. JAQL can read directly from http with the _____
   a. HTTP io adapter
   b. FTP adapter
   c. TCP adapter
   d. SOAP adapter

4. _____provide a way to package Jaql code in a re-usable fashion
   a. Libraries
   b. Modules
   c. Classes
   d. Functions

Checkpoint solution (2 of 2)

3. JAQL can read directly from http with the _____
   a. HTTP io adapter
   b. FTP adapter
   c. TCP adapter
   d. SOAP adapter

4. _____provide a way to package Jaql code in a re-usable fashion
   a. Libraries
   b. Modules
   c. Classes
   d. Functions

Unit summary

Having completed this unit, you should be able to:

- Understand overview of Jaql
- Understand basics of Jaql Language
- Understand Core Operators of Jaql Language
- Understand SQL Support for Jaql Language
- Understand MapReduce implementation in JAQL
- Understand Jaql I/O Systems