NDBI040: PRACTICAL CLASS 8

SCIDB

Tutor: Pavel Čontoš; December 9th 2020

(RECOMMENDED) REQUIREMENTS

- Database concepts

- SciDB 18.1 + Ubuntu 14.04 Image (link on practical class website)
- VirtualBox or VMWare Fusion / Workstation Player
  - Use port forwarding and ssh between host and guest, at your convenience

- macOS / Linux command line or PuTTy / WinSCP on Windows

SERVER ACCESS

CONNECT TO NOSQL SERVER

- `ssh` on macOS / Linux
- `PuTTy` on Windows

- nosql.ms.mff.cuni.cz:42222
- Login and password send by e-mail
- Change your initial password (if not yet changed) by `passwd`

TRANSFER FILES

- `scp` on macOS / Linux
- `WinSCP` on Windows

SCIDB

- Open source
- Array database
- Shared-nothing architecture

- High performance operations on ordered data
  - Spatial (location-based) data,
  - Temporal (time series) data,
  - Matrix-based data for linear algebra operations

- ACID transactions with versioned arrays
  - Array-level locking - lock acquired at the beginning of a transaction and released upon completion of the query
  - Write transactions may create new version of the array rather than modify existing array

ARRAY DATA MODEL

- Database → Array → Dimension → Cell → Attribute

- `Array`
  - Multidimensional array having specified dimensions and attributes
  - Has a unique name within the database
  - The schema of an array contains array attributes and dimensions

- `Dimension`
  - Consists of a list of index values
  - The number of index values is equal to dimension size
  - Divided into chunks, uniformly distributed using a round-robin
- `Cell`
  - May contain multiple attributes
- `Attribute`
  - Contains data

QUERY LANGUAGE

ARRAY QUERY LANGUAGE (AQL)

- Declarative language similar to SQL
- Includes data loading, selection and projection, aggregation and joins
- DDL statements define arrays and load data, DML statements access and operate on array data

ARRAY FUNCTIONAL LANGUAGE (AFL)

- Functional language
- Uses operators to compose queries or statements
- Operators allow data processing and aggregation, data exchange and storage

FIRST STEPS

- `scidb.py initall databaseName`
  - Initializes SciDB on the server
- `scidb.py startall databaseName`
  - Starts local SciDB instance
- `scidb.py status databaseName`
  - Reports the status of the various instances
- `scidb.py stopall databaseName`
  - Stops all SciDB instances

- `scidb.py startall mydb`
- `scidb.py stopall mydb`

IQUERY

- Default and interactive Linux shell interface that supports AQL and AFL statements
- By default, opens an AQL command prompt

INTERACTIVE MODE

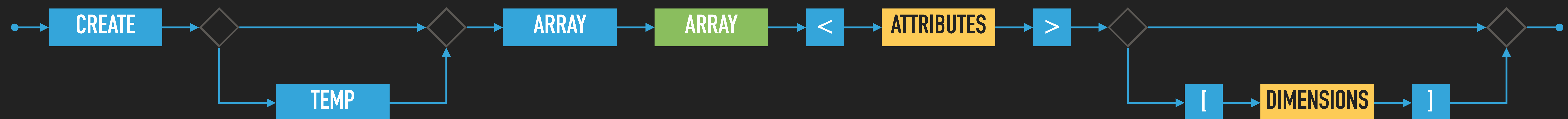
- `set lang AFL | AQL;` switches to AFL/AQL queries
- `help` displays commands reference

COMMAND LINE MODE

- `iquery -q "statement"` passes AQL query directly from command line
- `iquery -aq "statement"` passes AFL query directly
- `iquery -f "filename"` passes a file containing AQL statements
- `iquery -af "filename"` passes a file containing AFL statements
- `iquery -r "filename"` redirects the output to a file, otherwise prints result to stdout

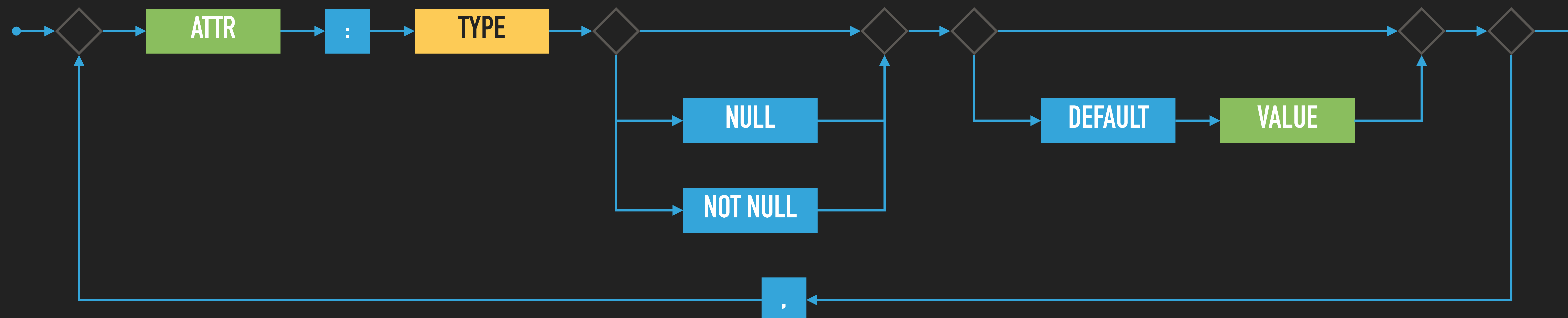
CREATE ARRAY

- `Array`
- `Temporary Array`
  - May improve performance but does not offer ACID transactions
  - Not saved to disk (not persistent)
  - Does not have versions, i.e. updates overwrite existing attribute values
  - Must be deleted explicitly, otherwise marked as unavailable after SciDB restart
- `DataFrame`
  - An array whose dimension do not have to be specified (managed implicitly)
  - Unordered collection of cells



ATTRIBUTES

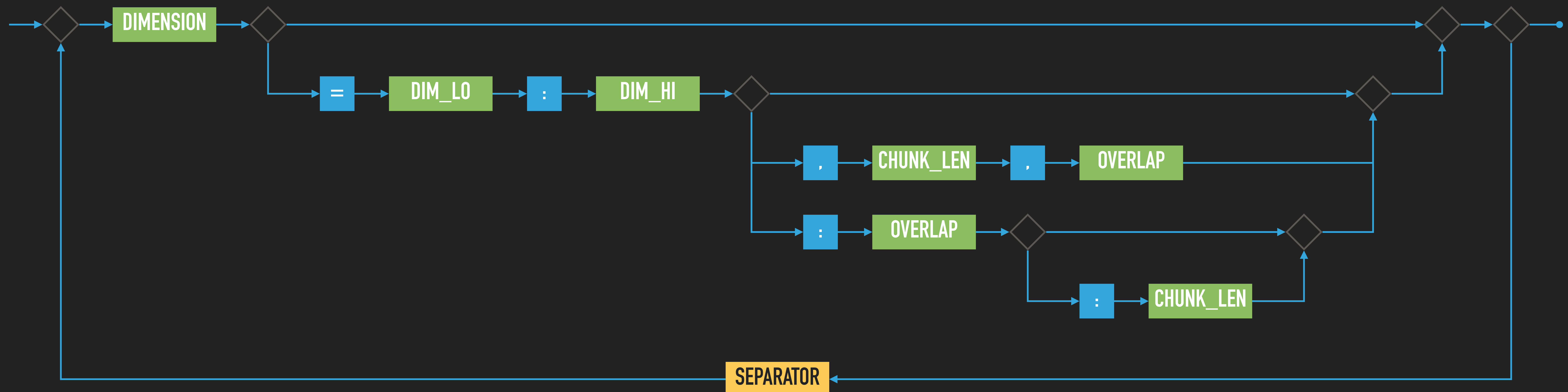
- Contain the actual data
- No duplicit attribute names allowed in the same array
- Use `list('types')` to see the list of available types
- `NULL` - attribute may contain null value, default value used otherwise
- `DEFAULT` - default value replacement for null if null not allowed



(UNIQUELY NAMED) DIMENSIONS

- `dim_lo` - the starting coordinate of a dimension
- `dim_hi` - the ending coordinate of a dimension, * if unbounded
- `chunk_len` - number of values per chunk
- `overlap` - number of overlapping values for adjacent chunks

- Attribute or dimension?
  - Dimensions form a coordinate system for an array
  - Adding dimensions to an array improves the performance of many types of queries by speeding up access to array data
  - Dimensions may be non-integer, i.e. `[ID(string)]`



EXAMPLE: CREATE ARRAY

- Creates one-dimensional arrays that represents actors, theirs roles and movies

- `CREATE ARRAY actors <actor:string, name:string, surname:string, year:int16> [i=0:*];`

ARRAY-RELATED AFL OPERATORS

- `list('arrays', false);` lists all arrays in the database
- `show(actors);` displays an array schema, equal to `SELECT * FROM show(actors);`
- `scan(actors);` displays an array content, equal to `SELECT * FROM actors;`
- `project(actors, actor, name, surname);` projects the data, equal to `SELECT actor, name, surname FROM actors;`
- `rename(actors, newActors);` renames an array, similar to `SELECT * INTO newActors FROM actors;`

EXERCISE 1

- Create arrays for movies and roles and set appropriate dimensions
  - movies
    - identifier: string, title: string, year: int16, rating: int8, length: int16
    - dimension j=0:3
  - roles
    - actor: string, role: string, movie: string, award: string
    - dimension k=0:*

LOADING AND SAVING DATA

- Before loading data, you must have created an array to load your data into
- SciDB proprietary format required
- Missing values are either substituted by default value (`0`, `""`) or `null` is used when allowed

- `LOAD array FROM 'path';`
  - AQL statement to load the data from file into the array
- `load(array, 'path)';`
  - AFL operator to load the data
- `SAVE array INTO 'path';`
  - AQL statement to save the data from the array into the file
- `save(array, 'path');`
  - AFL operator to save the data

```
[ ("trojan", "Ivan", "Trojan", 1964),
  ("machacek", "Jiri", "Machacek", 1966),
  ("schneiderova", "Jitka", "Schneiderova", 1973),
  ("sverak", "Zdenek", "Sverak", 1936) ]

[ ("vratnelahve", "Vratne lahve", 2006, 76, 99),
  ("samotari", "Samotari", 2000, 84, 103),
  ("medvidek", "Medvidek", 2007, 53, 100),
  ("stesti", "Stesti", 2005, 72, 100) ]

[ ("machacek", "Robert Landa", "vratnelahve", null),
  ("sverak", "Josef Tkaloun", "vratnelahve", null),
  ("trojan", "Ondrej", "samotari", null),
  ("machacek", "Jakub", "samotari", null),
  ("schneiderova", "Hanka", "samotari", null),
  ("trojan", "Ivan", "medvidek", null),
  ("machacek", "Jirka", "medvidek", "Czech Lion") ]
```

EXAMPLE: LOADING DATA

- Download `actors.scidb`, `movies.scidb` and `roles.scidb` data files from practical class website and load them into appropriate arrays

- AQL:
  - `LOAD actors FROM '/home/scidb/actors.scidb';`
  - `LOAD movies FROM '/home/scidb/movies.scidb';`
  - `LOAD roles FROM '/home/scidb/roles.scidb';`
- AFL:
  - `load(actors, '/home/scidb/actors.scidb');`
  - `load(movies, '/home/scidb/movies.scidb');`
  - `load(roles, '/home/scidb/roles.scidb');`

INSERT VALUE

- `insert(source_array, target_array);`
  - Inserts values from a source array into a target array
  - Rewrites or adds a value into target array depending on existence of a value in the target array
  - Equivalent AQL statement is `INSERT INTO sourceArray targetArray`

- `store(operator(operator_args), target_array);`
- `store(source_array, target_array);`
  - Saves the result from operator(operator_args) into an existing or new target array
  - Duplicates an array

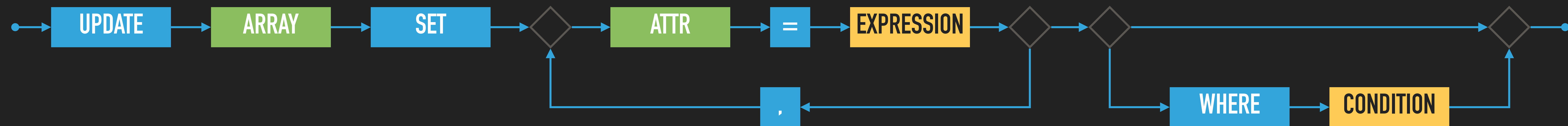
- `store(filter(actors, year >= 1966), youngActors);`

EXERCISE 2

- Insert the following data into array of actors
  - identifier: geislerova
  - name: Anna
  - surname: Geislerova
  - year: 1976
- Do not rewrite any existing actor

UPDATE VALUE

- When an array is updated, a new array version is created
  - "no overwrite" storage model



- `UPDATE movies SET rating = rating +10, length = length - 20 WHERE rating < 70;`

- You can list versions and browse the contents of any previous versions by using the version number or the array timestamp
  - `SELECT * FROM versions(actors);`
  - `list('arrays', true);`
  - `SELECT * FROM actors@1;`
  - `scan(actors@datetime('...'));`

APPLY OPERATOR

- `apply(array, newAttribute1, expression1[, ..., ..., newAttributeN, expressionN]);`
    - Produces an array with an additional attributes
    - The schema of the resulting array is modified (added attribute(s))
    - The shape of the resulting array is the same

- `apply(actors, fullname, name + ' ' + surname);`

EXERCISE 3

- Use apply operator to derive age of actors
  - Store result into a new array, i.e. use operator store

DELETE VALUE, REMOVE ARRAY

- `delete(array, expression);`
  - Deletes data from an array that satisfy an expression

- `remove(array);`
  - AFL statement that removes an array including all of its versions and its schema definition
  - Equivalent AQL statement is `DROP ARRAY array`

- `remove_versions(array, version_id);`
- `remove_versions(array, keep: count);`
- `remove_versions(array);`

QUERY

- AQL's Data Manipulation Language (DML) provides queries to access and operate on data

QUERY STRUCTURE

- **Data types**
  - Define the classes of values that database can store and perform operations on
  - `list('types');` lists all allowed types
- **Operators**
  - Accepts one or more array as an input and return an array as output
  - May be used as a standalone or nested within AFL, or within AQL
  - `list('operators');` lists all allowed operators
- **Functions**
  - Accepts scalar value or one/more arrays as arguments and return a scalar value
  - `list('functions');` lists all allowed functions, i.e. comparison functions
- **Aggregates**
  - Take an arbitrarily set of values as its input and outputs single scalar value
  - `list('aggregates');` lists all allowed aggregates functions

ARRAY JOINS

- `join(leftArray, rightArray);`
    - I.e. combines the attributes of two input arrays at matching dimension values
    - Equivalent AQL statement is `SELECT * FROM leftArray, rightArray;`
- `merge(leftArray, rightArray)`
    - Requires both arrays have the same number and types of attributes
    - Merges data from two arrays
    - Equivalent AQL statement is `SELECT * FROM merge(leftArray, rightArray);`
- `cross_join(leftArray [AS leftAlias], rightArray [AS rightAlias], [leftAlias.]leftDim1, [rightAlias.]rightDim1, ...);`
    - Provides a cross-product join of two arrays
    - Dimensions match by explicitly provided pairings
    - Array operands may have unmatched dimensions
    - Equivalent AQL statement is `SELECT * FROM cross_join(leftArray, rightArray, leftDim1, rightDim1, ...);`

ARRAY JOINS

- `JOIN ... ON` statement
  - Calculates the multidimensional join of two arrays after applying the constraints specified in the ON clause
  - The result is a subset(eq) of a `cross_join()`
  - Duplicit attributes are renamed by suffix convention

- `SELECT * FROM actors, roles;`
- `SELECT * FROM cross_join(actors, roles);`
- `SELECT * INTO actorsRoles FROM actors JOIN roles ON actors.actor = roles.actor;`

FILTERING OPERATORS

- `project(array, attribute, ...);`
  - Projects a subset of attributes from a source array
  - Equivalent AQL statement is `SELECT attribute, ... FROM array;`
- `filter(array, expression);`
  - Filters out values based on a boolean expression
  - Regular expressions may be used, i.e. `filter(list('operators'),regex(name,'(.*)q(.*)'));`
  - Equivalent AQL statement is `SELECT * FROM array WHERE expression;`
- `between(array, lowCoord1[, ..., lowCoordN], highCoord1[, ..., highCoordN]);`
  - Produces a subarray that is specified by a list of coordinates of an input array
  - HighCoordN does not have to be set, i.e. `null` value is allowed
  - Equivalent AQL statement is `SELECT * FROM between(array, lowCoord1[, ..., lowCoordN], highCoord1[, ..., highCoordN]);`

FILTERING OPERATORS

- `slice(array, dimension1, value1[, ..., ..., dimensionN, valueN]);`
  - Produces an array that is a subset of the source array where one or more dimension values is constant
  - Equivalent AQL statement is `SELECT * FROM array WHERE dimension1 = value1, ..., dimensionN = valueN;`

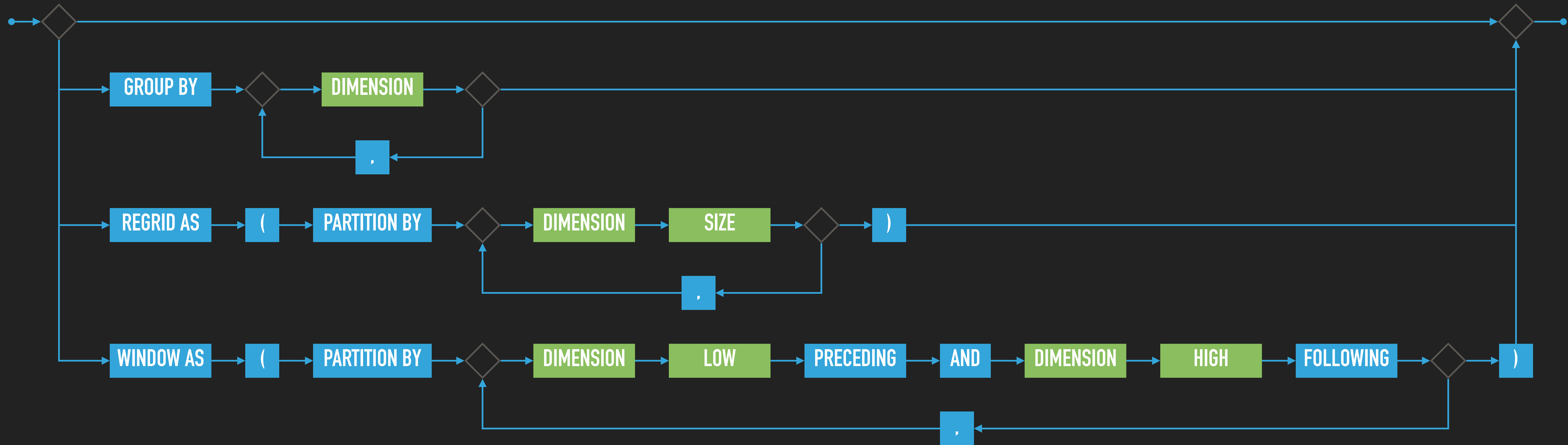
- `subarray(array, lowCoord1[, ..., lowCoordN], highCoord1[, ..., highCoordN]);`
  - Produces a subarray whose shape is defined by the boundary coordinates
  - The `between()` operator is similar, except that it returns an array with the same shape as the input array
  - `SELECT * FROM subarray(array, lowCoord1[, ..., lowCoordN], highCoord1[, ..., highCoordN]);`

EXERCISE 4

- Rewrite following AQL queries into equivalent AFL statements

- `SELECT name, surname FROM actors WHERE name = 'Ivan';`
- `SELECT * FROM between(actors, 0, 1);`
- `SELECT * FROM actorsRoles WHERE i=0;`
- `SELECT * FROM actorsRoles WHERE k=3;`

AGGREGATING OPERATORS AND FUNCTIONS



- `Grand aggregates` compute summaries over entire arrays
- `Group-by aggregates` compute summaries by grouping array data by dimension values
- `Grid aggregates` compute summaries for non-overlapping subarrays
- `Window aggregates` compute summaries over a moving window in an array

AGGREGATING OPERATORS AND FUNCTIONS

OPERATORS

- `aggregate(array, aggregateFn1(attribute)[as Alias1][, ...] [, dimension1, ...]);`
- `regrid(array, grid1[, ...], aggregateFn1(attribute)[ as Alias1][, ...]);`
- `window(array, dimPre1, dimFol1[,...], aggregateFn1(attribute)[ as Alias1][, ...]);`

FUNCTIONS

- `count(attribute | *)`, `approxdc(attribute)`
- `avg(attribute)`, `var(attribute)`, `stddev(attribute)`
- `min(attribute)`, `max(attribute)`
- `sum(attribute)`, `prod(attribute)`

EXAMPLE: AGGREGATING OPERATORS AND FUNCTIONS

- `SELECT avg(year), count(award), count(*), min(year), max(year), sum(year) FROM actorsRoles;`
- `aggregate(actorsRoles, avg(year), count(award), count(*), min(year), max(year), sum(year));`

- `SELECT max(year) FROM actorsRoles GROUP BY i;`
- `aggregate(actorsRoles, max(year), i);`

- `SELECT sum(year) AS sumYear FROM actorsRoles REGRID AS (PARTITION BY i 2, k 2);`
- `regrid(actorsRoles, 2, 2, sum(year) AS sumYear);`

- `SELECT sum(year) FROM actorsRoles WINDOW AS (PARTITION BY i 1 PRECEDING AND 3 FOLLOWING, k 1 PRECEDING AND 2 FOLLOWING);`
- `window(actorsRoles, 1, 3, 1, 2, sum(year));`

EXERCISE 5: NESTED SUBQUERIES

- Rewrite following nested AQL queries into AFL statements

- ```
  SELECT min(actor) AS actor, count(*), count(award) FROM (SELECT * FROM actorsRoles
  WHERE year > 1964) GROUP BY i;
  ```

- ```
  SELECT min(actor) AS actor, count(*), count(award) FROM (SELECT * FROM actorsRoles
  WHERE year > 1964 ORDER BY actor DESC) GROUP BY n;
  ```

REFERENCES

- SciDB Reference Guide
  - https://paradigm4.atlassian.net/wiki/spaces/scidb/pages/730268216/SciDB+Reference+Guide
- SciDB Operators
  - https://paradigm4.atlassian.net/wiki/spaces/scidb/pages/730268277/SciDB+Operators
- SciDB Functions
  - https://paradigm4.atlassian.net/wiki/spaces/scidb/pages/730269046/SciDB+Functions