Modern Database Systems

Other types of modern (not only) database systems

Doc. RNDr. Irena Holubova, Ph.D.
Irena.Holubova@matfyz.cuni.cz

Modern Data Management Systems

- NoSQL databases
  - Non-core – XML, object, …
  - Core – key/value, column, document, graph
- Multi-model databases and polystores
- NewSQL databases
- Array databases
- Search engines
  - Elasticsearch, Splunk, Solr, …
- …
- And there is also a number of specialized DBMSs
  - Navigational, multi-value, event, content, time-series, ...

NewSQL Databases

NewSQL Databases

- Idea (from 2011): scalable storage + all functionality known from traditional relational databases
  - Not just SQL access, but classical relational model, ACID properties, …
  - Previously ScalableSQL

Aslett, M.: *What We Talk about When We Talk about NewSQL*. 452 Group, 2011. http://blogs.the451group.com/information_management/2011/04/06/what-we-talk-about-when-we-talk-about-newsql/

Stonebraker, M.: *New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps*, 2011. https://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext

NewSQL Databases

http://www.cs.brown.edu/courses/cs227/slides/newsql/newsql-intro.pdf

- Approaches:
  - Distributed systems which add advantages of relational model + ACID
    - e.g. Clustrix, ScaleArc, MemSQL, VoltDB, …
  - Relational DBMSs extended towards horizontal scalability
    - e.g. TokuDB, JustOne DB, ..
- Cloud: NewSQL as a Service
  - Special type of a cloud service = scalable relational DBMS
    - e.g. Amazon Relational Database Service, Microsoft Azure Database, …

NewSQL Databases

- Why do we need them?
  1. There are applications which work with relational databases + they need to solve new increase of data volumes
     - Transformation to any NoSQL data model would be too expensive
  2. There are application which still need strong data consistency + horizontal scalability
- Consequence: Again NewSQL does not mean the end of traditional SQL (relational) DBMSs
  - An alternative approach – we need alternatives and there will occur other

↓

Stonebraker, M. et al.: *The end of an architectural era: (it's time for a complete rewrite)*. VLDB '07.

VOLTDB

- Based on academic DBMS H-System
  - Developed by researchers from US top universities (including M. Stonebraker) + Intel
  - Aim: relational model + ACID + horizontal scalability
- User perspective: classical relational DBMS
  - `CREATE / ALTER / DROP TABLE`, `INSERT INTO`, `CHECK` constraints, `SELECT` (including `GROUP BY`), set operations, nested queries, stored procedures, database views, …
- Big Data
  - Automatic data distribution
    - Users can specify according to which column to distribute
      - Customers: cities, countries, type, …
  - Shared-nothing architecture
    - Nodes in the cluster do not share memory, disk space, …
    - Autonomous parts which communicate using messages

https://www.voltdb.com/

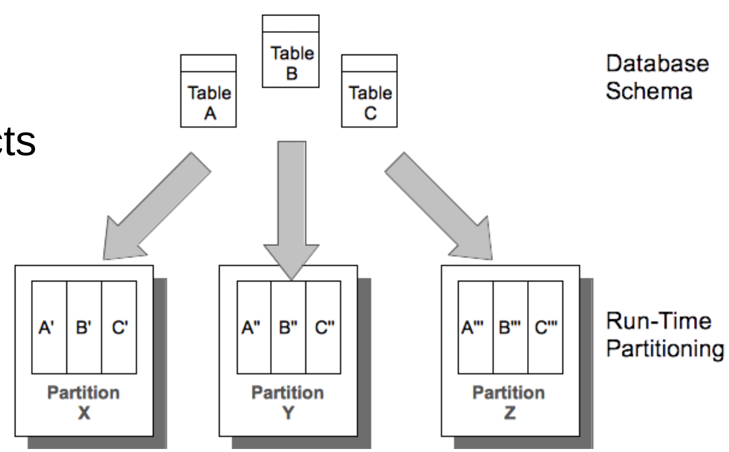
VOLTDB and research behind

- Observation: Traditional databases spend less than 10% of their time doing actual work
- Most of the time they focus on:

1. Page Buffer Management
   - Assigns database records to fixed-size pages, organizes their placement within pages, manages which pages are loaded into memory / are on disk, tracks dirty / clean pages as they are read and written to, …
2. Concurrency Management
   - Multiple user transactions operating concurrently must not conflict and must read consistent data
   - Database software has multiple threads of execution = data structures must be thread safe

VOLTDB

- In-memory database
  - Data are primarily processed in memory
    - Durability: command log (enterprise edition) / snapshots (community edition)
  - Eliminating disk waits
- All data operations in VoltDB are single-threaded
  - Simple data structures
  - Eliminating thread safety or concurrent access costs
- Distributed data processing
  - Includes distribution of stored procedures
    - Thanks to an analysis and pre-compilation of the data access logic in the procedures
    - Procedures work with local part of the data in separate transactions
      - 1 stored procedure = 1 transaction
  - Local transactions are serialized = no conflicts
    - No need for locks etc.
  - Distributed data processing works in parallel

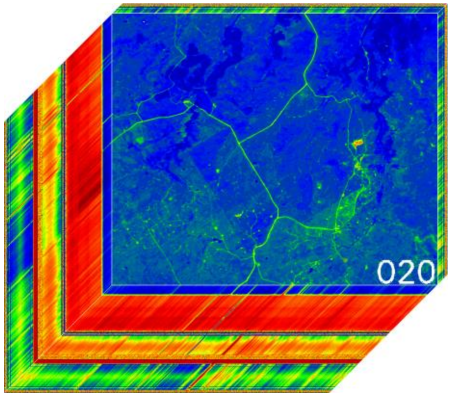


VOLTDB

- Replication
  - Partitions: peer-to-peer
  - Whole databases: peer-to-peer or master/slave
- Each node in the cluster contains a unique "slice" of the data and the data processing
  - Data + stored procedures
- Processing:
  1. When a procedure works with data on a single node (partition): no requests for other nodes
     - They can handle other requests in parallel
  2. Need for data from multiple nodes (partitions):
     1. One node in the cluster becomes a coordinator
     2. It hands out the necessary work to the other nodes
     3. It merges the results and ends the procedure

Array Databases

Array Databases



- Database systems specific for data represented as one- or multi-dimensional arrays
- Usually: We need to represent the respective values in time and/or space
  - Biology, chemistry, physics, geology, …
  - Complex research analyses of natural events
    - e.g. astronomical measurements, changes of climate, satellite pictures of the Earth, oceanographic data, human genome, …
- Example: Each satellite picture is a 2D-array (longitude + latitude) with values informing about the particular positions
  - Next dimensions: time when the picture was taken, characteristics of the tool taking the picture, …

Array Databases

- In general:
  - Big Data of a specific type
  - Data not suitable for flat 2D relations
    - Some RDBMSs support arrays
    - Too simple operations for these purposes
      - Not efficient
- Examples: SciDB, Rasdaman, Oracle Spatial and Graph, …

SciDB

- Provided by paradigm4
  - Co-founder: M. Stonebraker
- One of the most popular representatives
  - Wide range of functionalities
- Data model
  - Multidimensional sorted array
  - Assumption: data are not overwritten
    - Update = creating a new version of data
    - Aim: analyses of evolution/errors/corrections/… in time

If not explicitly specified

http://www.paradigm4.com/

SciDB

- **AFL (Array Functional Language)**
- **AQL (Array Query Language)**
  - Inspired by SQL
  - Instead of tables we work with arrays
    - Wider set of operations for DDL, DML
  - Compiled into AFL

```
CREATE ARRAY A <x: double, err: double> [i=0:99,10,0,
   j=0:99,10,0];
LOAD A FROM '../examples/A.scidb';
```

- Each array has:
  - At least one attribute (`x`, `err`) with a datatype (2x `double`)
  - At least one dimension (`i`, `j`)
  - Each dimension has :
    - coordinates (`0-99`)
    - size of data chunks (`10` fields) and
    - eventual overlapping (`0`)

SciDB

- SciDB distributes the chunks of data
  - Not too big, not too small
  - Recommendation: 10-20 MB
    - Depending on the datatypes
- Coordinates do not have to be limited (*)
- Overlapping is optional
  - Suitable, e.g., for faster searching nearest neighbours
    - The data would probably be otherwise stored on another cluster node

```
// create two 1D arrays
CREATE ARRAY A <val_a:double>[i=0:9,10,0];
LOAD A FROM '../examples/exA.scidb';
CREATE ARRAY B <val_b:double>[j=0:9,10,0];
LOAD B FROM '../examples/exB.scidb';

// print values of coordinate i from array A
SELECT i FROM A;
[(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)]

// print values of attribute val_a from array A and val_b from
// array B
SELECT val_a FROM A;
[(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)]
SELECT val_b FROM B;
[(101),(102),(103),(104),(105),(106),(107),(108),(109),(110)]
```

SciDB

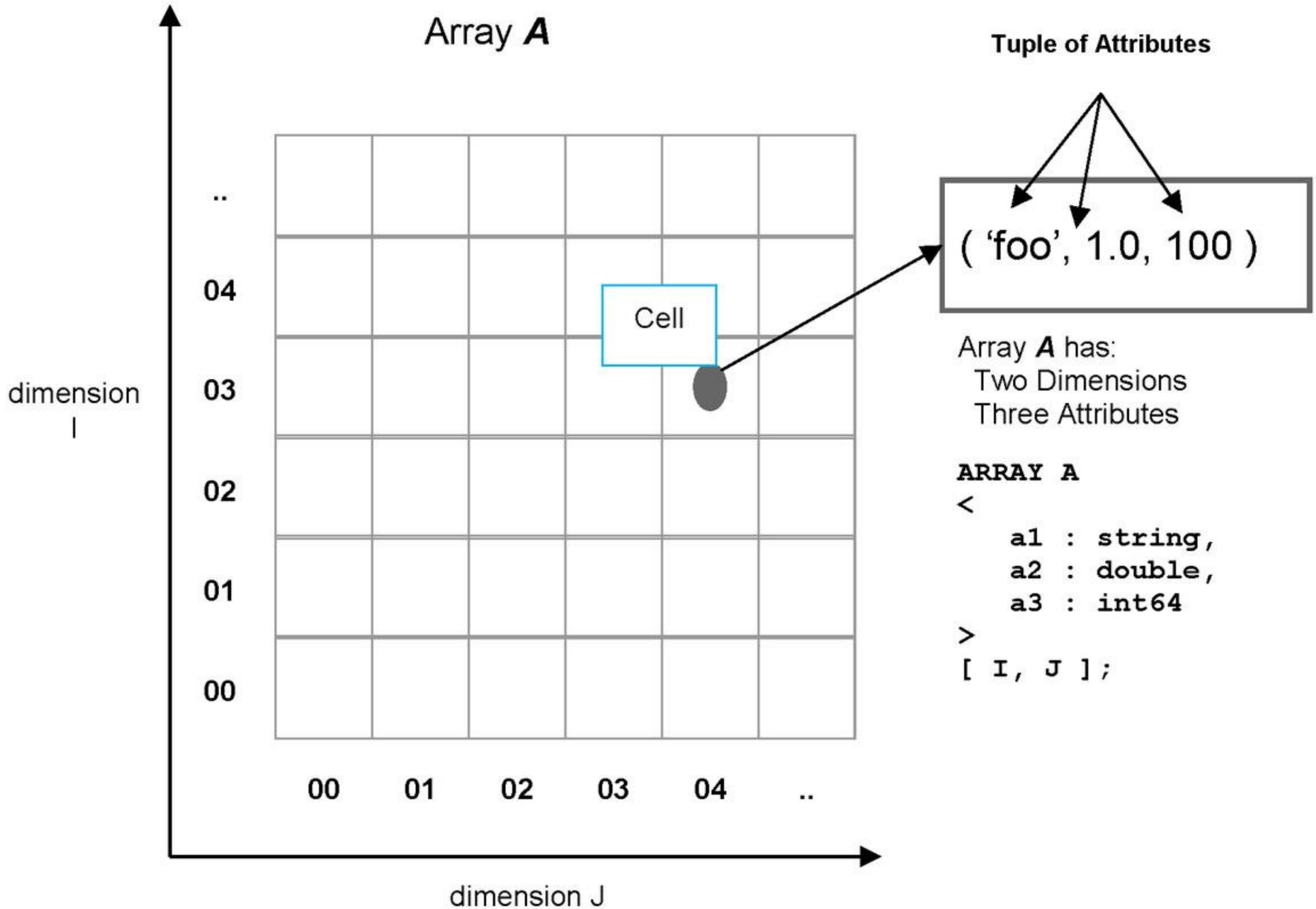
```
// usage of WHERE clause + sqrt() function
SELECT sqrt(val_b) FROM B WHERE j > 3 AND j < 7;
[(),(),(),(),(10.247),(10.2956),(10.3441),(),(),()]
```

- `SELECT` commands
  - Basic operation: inner join
    - Joined arrays must be compatible (coordinates, chunks, overlapping)
      - Amounts and datatypes of attributes can differ
      - Attributes are merged according to the given operation (condition)
  - Other joins: `MERGE`, `CROSS`, `CROSS_JOIN`, `JOIN ON` (a condition), …
  - Nested queries, aggregation (`GROUP BY`), sorting, …

```
// joining values of arrays A and B and storing to array C
SELECT * INTO C FROM A, B;
[(1,101),(2,102),(3,103),(4,104),(5,105),(6,106),(7,107),(8,108),(9,109),(10,110)]

// joining values of arrays C and B and storing to array D
SELECT * INTO D FROM C, B;
[(1,101,101),(2,102,102),(3,103,103),(4,104,104),(5,105,105),(6,106,106),(7,107,107),
(8,108,108),(9,109,109),(10,110,110)]

// print information about array D (see attributes with the same name)
SELECT * FROM show(D);
[("D<val_a:double,val_b:double,val_b_2:double> [i=0:9,10,0]")]

// joining the values by addition
SELECT C.val_b + D.val_b FROM C, D;
[(202),(204),(206),(208),(210),(212),(214),(216),(218),(220)]

// self-joining of values
SELECT a1.val_a, a2.val_a + 2 FROM A AS a1, A AS a2;
[(1,3),(2,4),(3,5),(4,6),(5,7),(6,8),(7,9),(8,10),(9,11),(10,12)]
```

More on Arrays

- Loosely based on *n*-dimensional matrices of linear algebra
- Each SciDB array consists of
  - Name
  - Ordered list of named dimensions
- Cell
  - Product of an array's dimensions
  - Record (tuple) of one or more named, typed, attributes
- Array's dimensions have a precedence order
  - E.g., array **B** is declared with dimensions **[x, y, z]**, **C** with the same dimensions in different order **[z, y, x]** => shape of **B** differs from **C**



Array ***A*** has:
Two Dimensions
Three Attributes

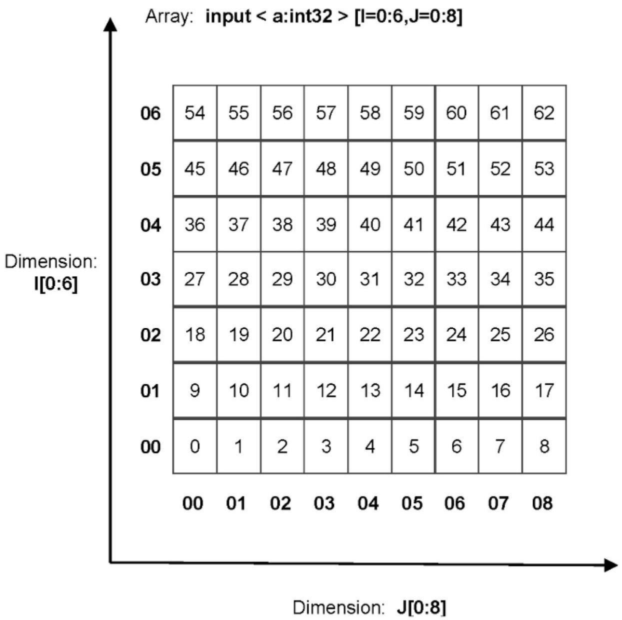
```
ARRAY A
<
    a1 : string,
    a2 : double,
    a3 : int64
>
[ I, J ];
```

More on Arrays

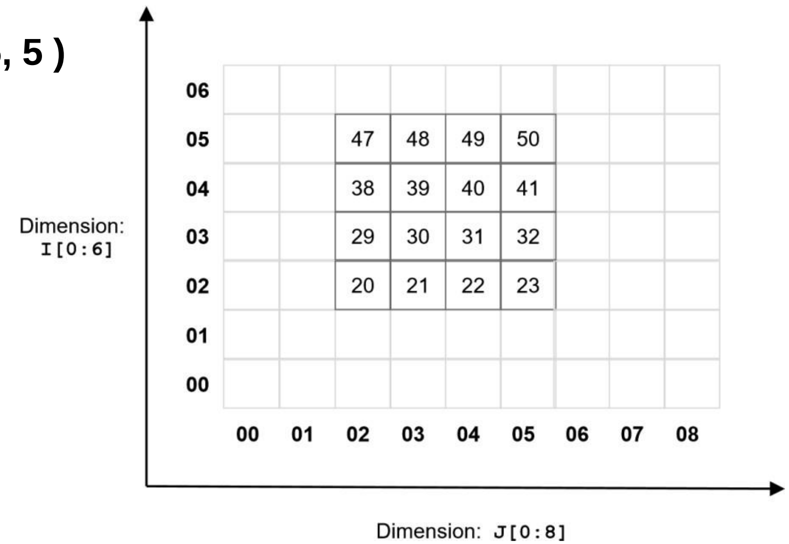
- SciDB arrays can either be sparse or dense
  - No internal distinction between them
  - Users can apply every operator to sparse or dense arrays
- SciDB can handle:
  - Dense data
    - e.g., images, mathematical matrices where every cell has value
  - Time series data
    - Typically with gaps in the series
  - Very sparse arrays
    - e.g. adjacency matrices to represent graphs
- Handling missing information
  - Specify a default value for an attribute or by using a missing code
  - Similar to the concept of a SQL null value
    - SciDB supports up to 128 codes = different kinds of missing-ness

Algebraic Operators

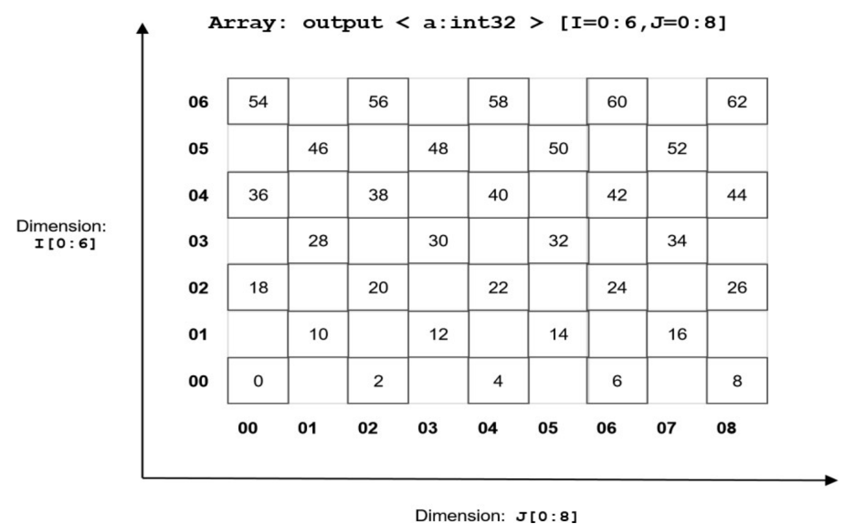
- Filter array data
- Calculate new values
- Combine data from multiple arrays
- Divide input arrays into partitions and compute various per-partition aggregates
  - Sum of values, centroid of a set of vectors, …
- Compute linear algebraic results
  - Matrix/matrix and matrix/vector multiply, array factorizations, image processing transformations, …
- …
- And they can be chained to form complex operations



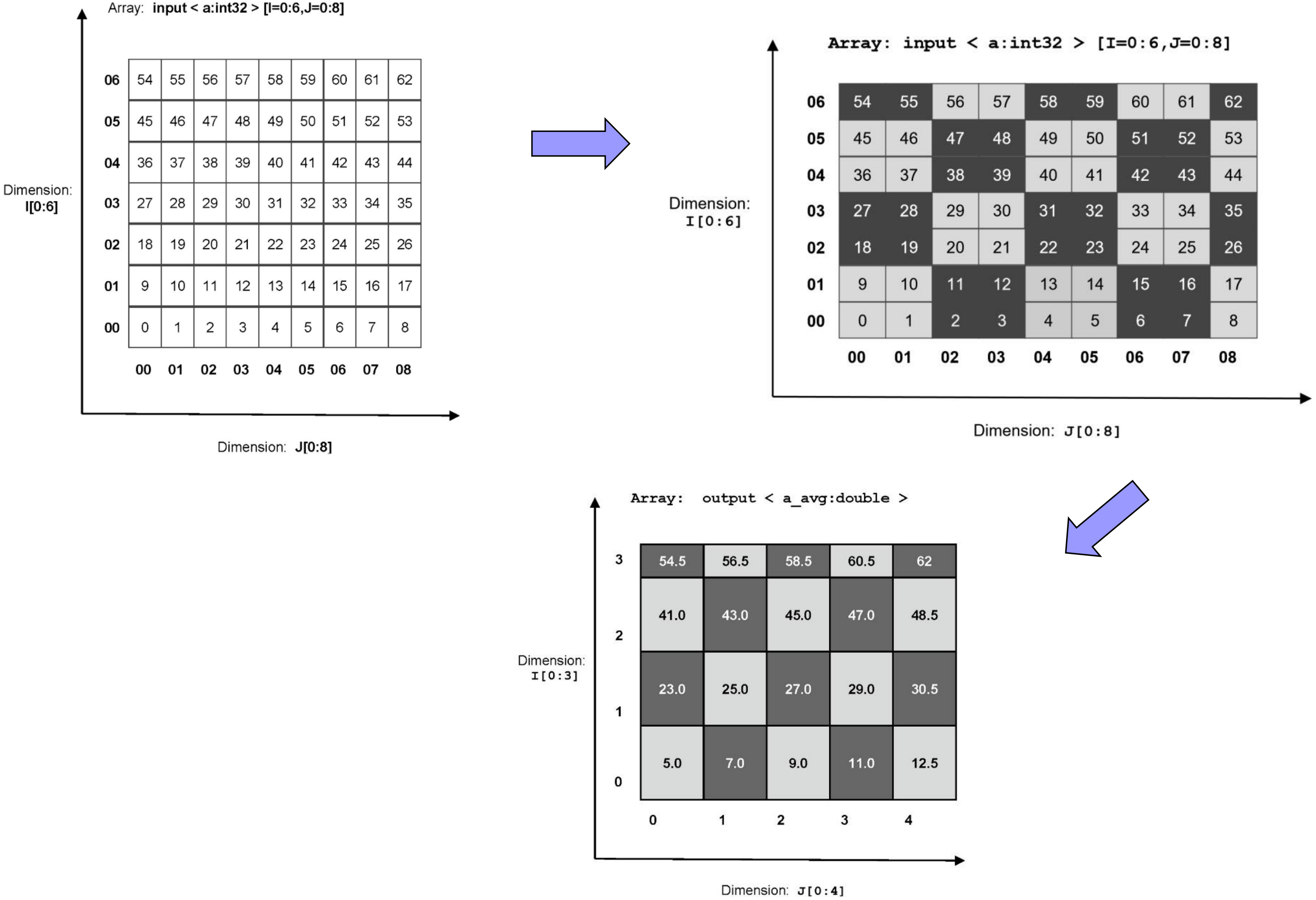
between (input, 2, 2, 5, 5)



filter (input, a%2 = 0)



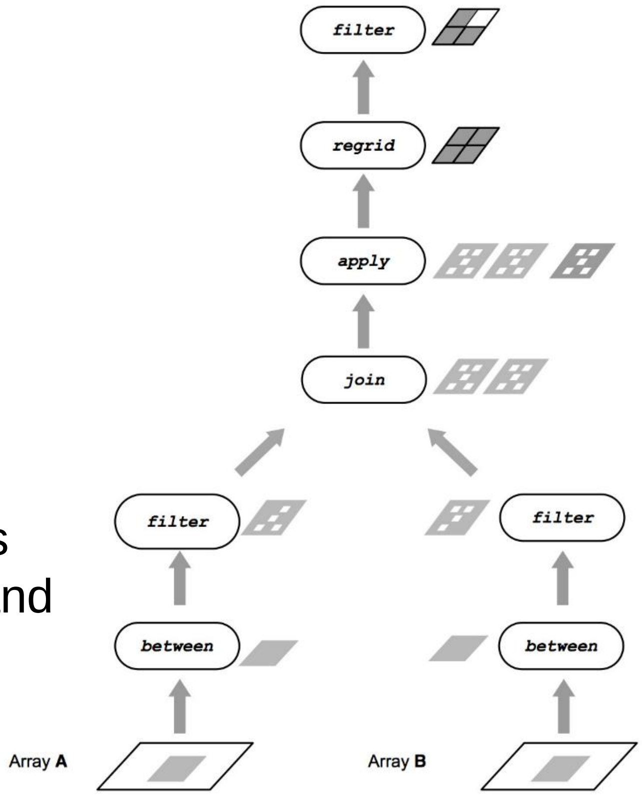
regrid (input, 2, 2, avg (a) as a_avg)



I \ J	00	01	02	03	04	05	06	07	08
06	54	55	56	57	58	59	60	61	62
05	45	46	47	48	49	50	51	52	53
04	36	37	38	39	40	41	42	43	44
03	27	28	29	30	31	32	33	34	35
02	18	19	20	21	22	23	24	25	26
01	9	10	11	12	13	14	15	16	17
00	0	1	2	3	4	5	6	7	8

I \ J	0	1	2	3	4
3	54.5	56.5	58.5	60.5	62
2	41.0	43.0	45.0	47.0	48.5
1	23.0	25.0	27.0	29.0	30.5
0	5.0	7.0	9.0	11.0	12.5

Query Evaluation



- Query = series of operators
- SciDB figures out an efficient, parallel execution strategy
  - Moves operators, injects new ones, replaces a particular sequence with a more efficient and logically equivalent alternative, …
- SciDB engine = data pump
  - Does not materialize intermediate results
    - Unless it is absolutely necessary
  - Passing data from the storage layer through a sequence of operators to compute the final result
  - Contrasts with the Map/Reduce model in Hadoop
    - Each link in a chain of Map/Reduce operations writes back to HDFS

Temporary Arrays

- Can improve performance
  - User-defined
- Do not offer the transactional guarantees of persistent arrays (ACID)
- Are not persistent (saved to disk)
  - In memory
- Become corrupted if a SciDB instance fails
  - When a SciDB cluster restarts, all temporary arrays are marked as unavailable
    - But not deleted; must be deleted explicitly
- Do not have versions
  - Any update overwrites existing attribute values

Array Attributes

- Store individual data values in array cells
- Consist of:
  - Name
  - Data type
  - Nullability (optional)
  - Default value (optional)
    - If unspecified, the system chooses a value:
      - If the attribute is nullable: null
      - Otherwise:
        - 0 for numeric types
        - empty string "" for string type
  - Compression type (optional): zlib or bzlib

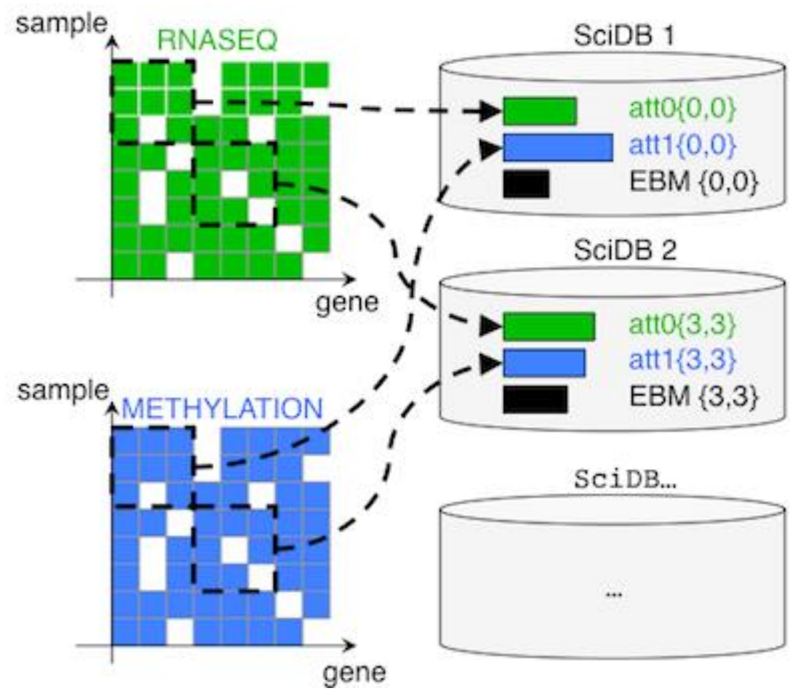
Datatype	Default value	Description
bool	false	Boolean value, true (1) or false (0)
char	\0	Single ASCII character
datetime	1970-01-01 00:00:00	Date and time
datetimetz	1970-01-01 00:00:00 -00:00	Date and time with timezone offset.
double	0	Double-precision floating point number
float	0	Single-precision floating-point number
int8	0	Signed 8-bit integer
int16	0	Signed 16-bit integer
int32	0	Signed 32-bit integer
int64	0	Signed 64-bit integer
string	''	Variable length character string, default is the empty string
uint8	0	Unsigned 8-bit integer
uint16	0	Unsigned 16-bit integer
uint32	0	Unsigned 32-bit integer
uint64	0	Unsigned 64-bit integer

Array Dimensions

- Form the coordinate system for a SciDB array
- Consist of:
  - Name
    - If only the name is specified: SciDB leaves the chunk length unspecified, uses the default overlap of zero, and makes the dimension unbounded.
  - Low value – dimension start value
  - High value – dimension end value (or * for no limit)
  - Chunk overlap (optional) – number of overlapping dimension values for adjacent chunks
  - Chunk length (optional) – number of dimension values between consecutive chunk boundaries
    - 1-dimensional array: maximum number of cells in each chunk
    - n-dimensional array: maximum number of cells in each chunk is the product of the chunk length parameters of each dimension

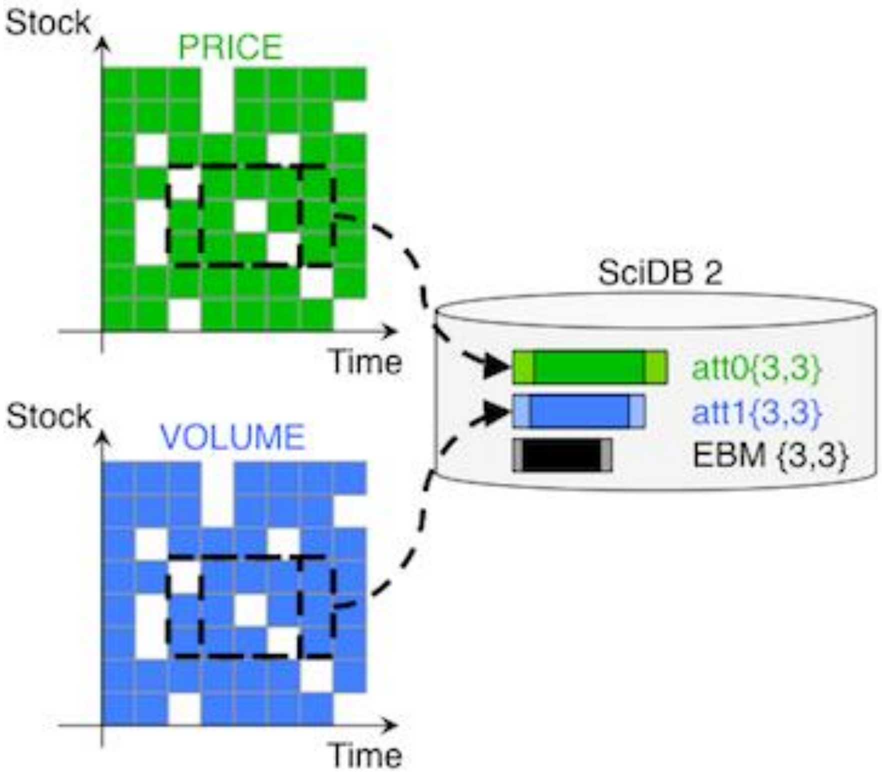
Multidimensional Array Clustering

- Makes sure that:
  1. Data that are close to each other in the user-defined coordinate system are stored in the same chunk
  2. Data are stored in the same order as in the coordinate system
- Different attributes are stored separately
- Data are split into rectilinear chunks
  - Chunks are assigned to different SciDB instances using a hash function
- Data in each chunk are stored in a contiguous region
- Data are compressed
- The locations of empty cells are encoded using a special bitmask EBM
- Coordinate values themselves are not stored, but are recomputed when needed from the EBM



Multidimensional Array Clustering

- Users can specify an optional overlap of chunks
  - Data in the overlap regions are replicated in the logically adjacent chunks
- Overlap is maintained automatically by the database
  - SciDB turns window queries into parallel operations that require no special programming on the part of the developer
- The overlap uses slightly more storage space but gives faster performance
  - To speed up windowed queries



SciDB Operators and Functions

- Operators:
  - https://paradigm4.atlassian.net/wiki/spaces/scidb/pages/2694414589/SciDB+Operators
- Functions:
  - https://paradigm4.atlassian.net/wiki/spaces/scidb/pages/2694416383/SciDB+Functions

Search Engines

Search Engines

- Sometimes denoted as search engine data management systems
- Differences from relational DBMSs
  - No rigid structural requirements
    - Data can be structured, semi-structured, unstructured, …
    - No relations, no constraints, no joins, no transactional behaviour, …
  - Use cases: relevance-based search, full text search, synonym search, log analysis, …
    - Not typical for databases
  - Data can be large
    - Distributed computing
- Differences from NoSQL DBMSs
  - Primarily designed for searching, not editing
  - Specialized functions: full-text search, stemming, complex search expressions, ranking and grouping of search results, geospatial search, …
    - Big Data analytics

elasticsearch

- Distributed full-text search engine
  - Scalable search solution
- Released in 2010
- Written in Java
  - Based on Lucene library
- HTTP web interface
  - JSON schema-free documents
- Official clients: Java, .NET (C#), PHP, Python, Apache Groovy, Ruby, …
- Elastic Stack = Elasticsearch +
  - Logstash – collects, processes, and forwards events and log messages
  - Kibana – analytics and visualization platform

logstash

kibana

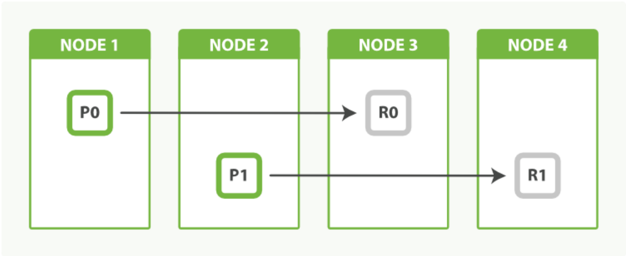
https://www.elastic.co/products/elasticsearch/

elasticsearch

- Can be used for all kinds of documents
- Near real-time search
  - Slight latency (approx. 1 second) from the time you index (or update or delete) a document until the time it becomes searchable
- Index = collection of documents with similar characteristics
  - e.g., customer data, product catalogue, …
  - Has a name
  - In a cluster there can be any number of indices
- Indices can be divided into shards
  - Each shard can have replicas
  - Rebalancing and routing are done automatically
- Each node can act as a coordinator to delegate operations to the respective shards

Indices

- When creating an index, define the number of shards and number of replicas
  - Note: index does not need to be defined beforehand
  - Each shard is in itself a fully-functional and independent index
  - Shards enable:
    - Horizontal scaling of large volumes of data
    - Parallelization of operations
  - Replicas enable:
    - High availability (partial failures)
    - Parallelization of operations
- Default: 5 primary shards and 1 replica
  - i.e., 10 shards per index



Basic Operations

```
GET /_cat/indices?v
```

- Get all indices

```
PUT /customer?pretty
```

- Create index "customer" (and pretty print the result, if any)

```
PUT /customer/_doc/1?pretty
{ "name": "John Doe" }
```

- Index the given document with ID = 1

```
GET /customer/_doc/1?pretty
```

- Get document with ID = 1

```
DELETE /customer/_doc/1?pretty
```

- Delete document with ID = 1

```
DELETE /customer
```

- Delete index "customer"

elasticsearch

Data Modification

- ID of a document
  - If an existing is used: the document is replaced (and re-indexed)
  - If a different is used: a new document is stored
    - The same one twice
  - If none is specified: a random ID is generated
- Document updates
  - No in-place updates
  - A document is deleted and a new one is created and indexed

Data Modification

```
POST /customer/_doc/1/_update?pretty
{ "doc": { "name": "Jane Doe" } }
```

- Change value of field “name” of document with ID = 1

```
POST /customer/_doc/1/_update?pretty
{ "doc": { "name": "Jane Doe", "age": 20 } }
```

- … and add a new field

```
POST /customer/_doc/1/_update?pretty
{ "script" : "ctx._source.age += 5" }
```

- … or use a script to specify the change

ctx._source = document content
ctx._index = document metadata
…

Batch Processing

```
POST /customer/_doc/_bulk?pretty
 {"index":{"_id":"1"}} {"name": "John Doe" }
 {"index":{"_id":"2"}} {"name": "Jane Doe" }
```

- Index two documents

```
POST /customer/_doc/_bulk?pretty
{"update":{"_id":"1"}}
  {"doc": { "name": "John Doe becomes Jane Doe" } }
{"delete":{"_id":"2"}}
```

- Update the first document, delete the second document

Search API

```
{ "account_number": 0,
  "balance": 16623,
  "firstname": "Bradshaw",
  "lastname": "Mckenzie",
  "age": 29,
  "gender": "F",
  "address": "244 Columbus Place",
  "employer": "Euron",
  "email": "bradshawmckenzie@euron.com",
  "city": "Hobucken",
  "state": "CO" }
```

- Sample data set

Search API

- Search parameters can be sent by:
  - REST request URI
  - REST request body
    - More expressive
    - More readable (JSON)?

```
GET /bank/_search?q=*&sort=account_number:asc&pretty
```

- Search (`_search`) in the `bank` index,
- match all the documents (`q=*`),
- sort the results using the `account_number` field of each document in an ascending order (`sort=account_number:asc`)

Search API

```
{
  "took" : 9,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1000,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
```

```
{
 "_index" : "holubova_bank",
 "_type" : "_doc",
 "_id" : "51",
 "_score" : 1.0,
 "_source" : {
   "account_number" : 51,
   "balance" : 14097,
   "firstname" : "Burton",
   "lastname" : "Meyers",
   "age" : 31,
   "gender" : "F",
   "address" : "334 River Street",
   "employer" : "Bezal",
   "email" : "burtonmeyers@bezal.com",
   "city" : "Jacksonburg",
   "state" : "MO"
 }
}, …
```

Search API

- In the result we will see:
  - `took` – time in milliseconds to execute the search
  - `timed_out` – if the search timed out or not
  - `_shards` – how many shards were searched
    - Total, successful, failed, skipped
  - `hits` – search results
    - `hits.total` – total number of documents matching our search criteria
    - `hits.hits` – actual array of search results
      - Default: first 10 documents
    - `hits.sort` – sort key for results
  - …

Search API

```
GET /bank/_search
{ "query": { "match_all": {} },
  "sort": [ { "account_number": "asc" } ] }
```

- The same exact search using the request body method

- When all search results are returned, Elasticsearch does not maintain any kind of server-side resources or open cursors etc.
  - Contrary to, e.g., traditional relational databases

Query DSL

- Domain specific language
- JSON-style

```
GET /bank/_search
{ "query": { "match_all": {} },
  "from": 10,  // starting index
  "size": 10,  // number of results
  "_source": ["account_number", "balance"]
               // include to the result
  "sort": { "balance": { "order": "desc" } }
 }
```

https://www.elastic.co/guide/en/elasticsearch/reference/6.5/query-dsl.html

Query DSL

```
"query": { "match": { "account_number": 20 } }
```

- Return the account numbered 20

```
"query": { "match": { "address": "mill" } }
```

- Return all accounts containing the term "`mill`" in the address

```
"query": { "match": { "address": "mill lane" } }
```

- Return all accounts containing the term "`mill`" or "`lane`" in the address

```
"query": { "match_phrase": { "address": "mill lane"
  } }
```

- Return all accounts containing the phrase "`mill lane`" in the address

Query DSL – Bool Query

- Bool query allows us to compose smaller queries into bigger queries using Boolean logic

```
"query": { "bool":
  { "must": [
    { "match": { "address": "mill" } },
    { "match": { "address": "lane" } } ] } }
```

- Return all accounts containing "`mill`" **and** "`lane`" in the address

```
"query": { "bool":
  { "should": [
    { "match": { "address": "mill" } },
    { "match": { "address": "lane" } } ] } }
```

- Return all accounts containing "`mill`" **or** "`lane`" in the address

Query DSL – Bool Query

```
"query": { "bool": {
  "must_not": [
    { "match": { "address": "mill" } },
    { "match": { "address": "lane" } } ] } }
```

- Return all accounts that contain **neither** "`mill`" **nor** "`lane`" in the address

```
"query": { "bool": {
  "must": [ { "match": { "age": "40" } } ],
  "must_not": [ { "match": { "state": "ID" } } ] } }
```

- Return all accounts of anybody who is 40 years old but doesn't live in ID(aho):

Query DSL – Filters

- `_score` field in the search results
  - Relative measure of how well the document matches the search query
    - The bigger, the more relevant
    - Practical scoring function evaluates it from 0 to `max_score` for the set
      - Idea: more relevant documents =
        - a) with a higher term frequency, and
        - b) contain more unique uses of the term compared to other documents in the index
  - When queries filter the set, it is not evaluated
    - Y/N depending on the filter

```
"query": {
  "bool": { "must": { "match_all": {} },
  "filter": {
    "range": { "balance": {
      "gte": 20000,
      "lte": 30000 } } } } }
```

- Return all accounts with balances between 20000 and 30000

Query DSL – Aggregations

- Ability to group and extract statistics
  - Like SQL GROUP BY
- We can execute searches returning both hits and aggregated results
  - No round tripping

```
GET /bank/_search {
  "size": 0,    // not show search hits
  "aggs": {
    "group_by_state": {
      "terms": { "field": "state.keyword" } } } }
```

- Group all the accounts by state, and returns the top 10 (default) states sorted by count descending (default)

Query DSL – Aggregations

```
GET /bank/_search {
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": { "field": "state.keyword" },
      "aggs": {
        "average_balance": {
          "avg": { "field": "balance" } } } } } }
```

- Calculate the average account balance by state
  - Uses nested aggregations (`average_balance` in `group_by_state`)

Apache Lucene

- Used by Elasticsearch, Solr, …
- Released 1999
- Written in Java
- High-performance, text search engine library
- Support for
  - Ranked searching
  - A number of query types: phrase queries, wildcard queries, proximity queries, range queries, …
  - Fielded searching
    - e.g. title, author, contents, …
  - Sorting by any field
  - Multiple-index searching with merged results
  - Simultaneous update and searching
  - Flexible faceting, highlighting, joins and result grouping

https://lucene.apache.org/core/

Apache Lucene

- Inverted index
- Document is the unit of search and index
  - Does not have to be real documents, but also, e.g., database tables
- Document consists of one or more fields
  - Name-value pair
- Searching requires an index to have already been built
- For searching it uses own language
  - Matching: keyword, wildcard, proximity, range searches, …
  - Logical operators
  - Boosting of terms/clauses
  - …

References

- VoltDB Documentation
  - https://docs.voltdb.com/
- SciDB Reference Guide
  - https://paradigm4.atlassian.net/wiki/spaces/scidb/overview?homepageId=2694289094
- Elastic Stack and Product Documentation
  - https://www.elastic.co/guide/index.html