2 Beyond Relational Database Systems

- Column Stores
- NoSQL Database Systems
- Graph-DB

2.1 Column Stores[1]

Table:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

row-wise storage:

A1	B1	C1	A2	B2	C2	A3	B3	C3

[1] see for more information:
`http://nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf`

2.1 Column Stores[1]

Table:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

row-wise storage:

A1	B1	C1	A2	B2	C2	A3	B3	C3

column-wise storage:

A1	A2	A3	B1	B2	B3	C1	C2	C3

[1] see for more information:
http://nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf

2.1 Column Stores[1]

Table:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

row-wise storage:

A1	B1	C1	A2	B2	C2	A3	B3	C3

column-wise storage:

A1	A2	A3	B1	B2	B3	C1	C2	C3

[1] see for more information:
`http://nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf`

Why Column-stores?

Advantages:

- Buffer management, only columns needed are read,
- data locality, column block iteration supports value aggregations,
- column-specific compression, values are from the same type,
- column insertion,
- late tuple materialization.

Disadvantage:

- Tuple insertion, and
- tuple reconstruction.

Compression and Suppression

Column compression

- Run-length encoding,
- Bit-vector encoding,
- Dictionary encoding,
- Reference encoding,
- Differential encoding.

Null suppression

- Position list encoding.

Run-length encoding

someAttribute
200
200
200
300
300
300
300
400
300
300

$\Longrightarrow$

someAttribute_compressed $(value, start, length)$
$(200, 1, 3)$
$(300, 4, 4)$
$(400, 8, 1)$
$(300, 9, 2)$

Bit-vector encoding

someAttribute
200
200
200
300
300
300
300
400
300
300

$\Longrightarrow$

someAttribute 200	*someAttribute* 300	*someAttribute* 400
1	0	0
1	0	0
1	0	0
0	1	0
0	1	0
0	1	0
0	1	0
0	0	1
0	1	0
0	1	0

Dictionary encoding

someAttribute
https : //www.informatik.uni – freiburg.de/
https : //wiki.dbpedia.org/
https : //de.wikipedia.org/wiki/World_Wide_Web
https : //www.informatik.uni – freiburg.de/
https : //wiki.dbpedia.org/
https : //de.wikipedia.org/wiki/World_Wide_Web
https : //www.uni – freiburg.de/

$\Longrightarrow$

someAttribute_encoded
1
2
3
1
2
3
4

Dictionary
1 → https : //www.informatik.uni – freiburg.de/
2 → https : //wiki.dbpedia.org/
3 → https : //de.wikipedia.org/wiki/World_Wide_Web
4 → https : //www.uni – freiburg.de/

Reference encoding

someAttribute		*someAttribute_compressed* *Reference* : 200
200		0
205		5
208		8
199		-1
195	$\Longrightarrow$	-5
300		#300
300		#300
200		0
210		10
205		5

Exeptions are indicated by #; necessary when difference to reference-value is larger then given range.

Differential encoding

someAttribute
200
205
208
199
195
300
300
200
210
205

$\Longrightarrow$

someAttribute_compressed   *Reference* : 200
0
5
3
-9
-4
#300
#300
5
10
-5

Exeptions are indicated by #; necessary when difference to previous value is larger then given range.

Null suppression: Position list encoding

someAttribute
NULL
https://wiki.dbpedia.org/
NULL
NULL
https://wiki.dbpedia.org/
NULL
https://www.uni-freiburg.de/
NULL
NULL

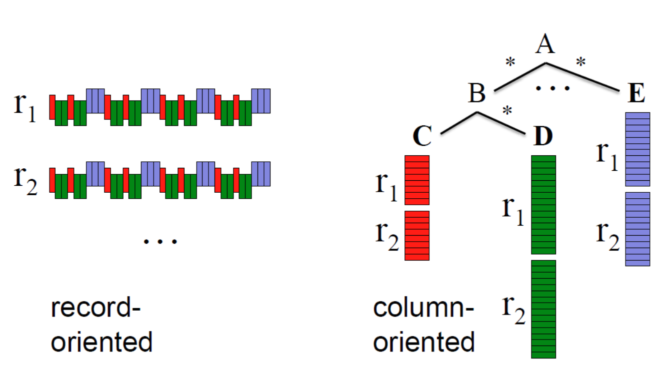
$\Longrightarrow$

someAttribute_encoded
2 $\rightarrow$ https://wiki.dbpedia.org/
5 $\rightarrow$ https://wiki.dbpedia.org/
7 $\rightarrow$ https://www.uni-freiburg.de/

Metadata
total number of rows : 9
number of nonnull value : 3

Column striping

How to store nested records in a column-store?



Lit.: Dremel: Interactive Analysis of Web-Scale Datasets By Sergey Melnik et al., Communications of the ACM, Vol. 54 No. 6, Pages 114-123
Dremel made simple with Parquet,
`https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html`

Data model

Strongly-typed nested records with abstract syntax:

$$\tau = dom \mid \langle A_1 : \tau[* \mid ?], \ldots, A_n : \tau[* \mid ?] \rangle$$

where τ is an atomic type or a record type.

- Atomic types in *dom*: integers, floating-point numbers, strings, etc.
- Records consist of one or multiple fields. Field i in a record has a name A_i and an optional multiplicity label.
- Repeated fields ($*$) may occur 0 or more times in a record.
- Optional fields ($?$) may be missing from the record.
- Otherwise, a field is required, i.e., must appear exactly once.

Data model

Strongly-typed nested records with abstract syntax:

$$\tau = dom \mid \langle A_1 : \tau[* \mid ?], \ldots, A_n : \tau[* \mid ?] \rangle$$

where τ is an atomic type or a record type.

- Atomic types in *dom*: integers, floating-point numbers, strings, etc.
- Records consist of one or multiple fields. Field i in a record has a name A_i and an optional multiplicity label.
- Repeated fields $(*)$ may occur 0 or more times in a record.
- Optional fields $(?)$ may be missing from the record.
- Otherwise, a field is required, i.e., must appear exactly once.

Sample nested tree-shaped records and their schema.

r_1

```
DocId: 10
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

r_2

```
DocId: 20
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

Column-striped representation: one column for each path from the root to a leaf.

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

. . . Repetition level r and definition level d are used to be able to reconstruct the original nested record!

Column striping

```
DocId: 10                     r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 20                     r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Column striping

```
DocId: 10                  r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 20                  r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Record structure is captured by *repetition level* and *definition level*.

Repetition level:

The *repetition level* for a path denotes at which level in the path the last repetition occurred. Only repeated fields in the path are counted.

Allows to derive when to start a new list and at which level.

Definition level:

The *definition level* specifies how many fields in a path p that could be undefined (because they are optional or repeated) are present.
In other words, stores the level for which the field is null: from 0 at the root of the schema up to the maximum level for this column as can be derived from the schema.

- A field is defined: all its parents are defined too,
- a field is null: record the level at which it started being null.
- null values must be inserted in case no value for the entire path but only for a prefix of the path exists.

A required field is always defined and does not get a definition level.

Record structure is captured by *repetition level* and *definition level*.

Repetition level:

The *repetition level* for a path denotes at which level in the path the last repetition occurred. Only repeated fields in the path are counted.

Allows to derive when to start a new list and at which level.

Definition level:

The *definition level* specifies how many fields in a path p that could be undefined (because they are optional or repeated) are present.
In other words, stores the level for which the field is null: from 0 at the root of the schema up to the maximum level for this column as can be derived from the schema.

- A field is defined: all its parents are defined too,
- a field is null: record the level at which it started being null.
- null values must be inserted in case no value for the entire path but only for a prefix of the path exists.

A required field is always defined and does not get a definition level.

The *definition level* specifies how many fields in a path *p* that could be undefined (because they are optional or repeated) are present.

```
message ExampleDefinitionLevel {
    optional group a {
        optional group b {
            optional string c;
        }
    }
}
```

Value	Definition Level
`a: null`	0
`a: { b: null }`	1
`a: { b: { c: null } }`	2
`a: { b: { c: "foo" } }`	3 (actually defined)

The *definition level* specifies how many fields in a path *p* that could be undefined (because they are optional or repeated) are present.

```
message ExampleDefinitionLevel {
    optional group a {
        required group b {
            optional string c;
        }
    }
}
```

Value	Definition Level
`a: null`	0
`a: { b: null }`	Impossible, as `b` is required
`a: { b: { c: null } }`	1
`a: { b: { c: "foo" } }`	2 (actually defined)

The *repetition level* for a path denotes at which level in the path the last repetition occurred. Only repeated fields in the path are counted.

Schema:	Data: [[a,b,c],[d,e,f,g]],[[h],[i,j]]
`message nestedLists { repeated group level1 { repeated string level2; } }`	`{ level1: { level2: a level2: b level2: c }, level1: { level2: d level2: e level2: f level2: g } } { level1: { level2: h }, level1: { level2: i level2: j } }`

Schema:

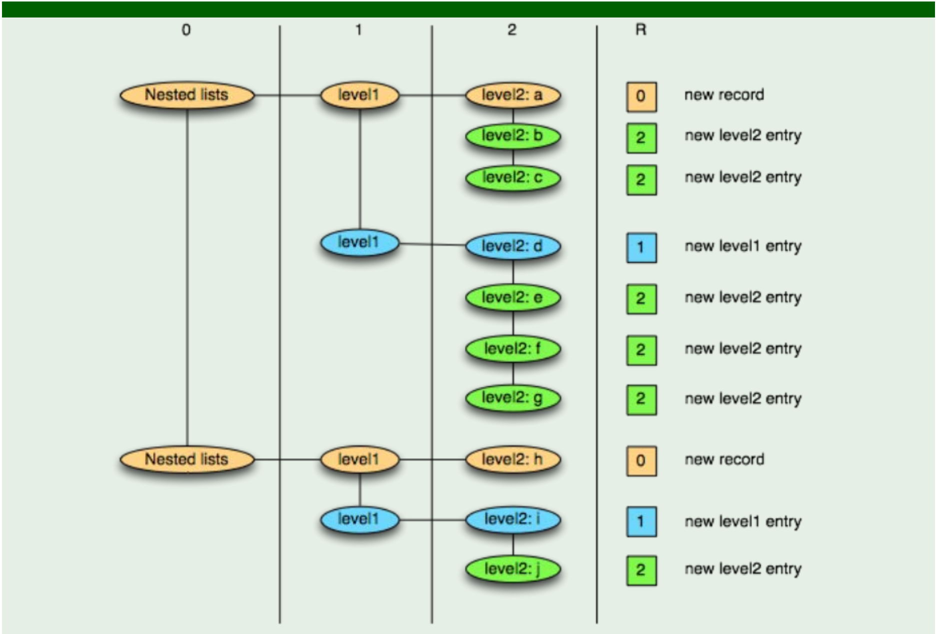
```
message nestedLists {
   repeated group level1 {
      repeated string level2;
   }
}
```

Data:

```
{
      level1: {
            level2: a
            level2: b
            level2: c
      },
      level1: {
            level2: d
            level2: e
            level2: f
            level2: g
      }
}
{
      level1: {
            level2: h
      },
      level1: {
            level2: i
            level2: j
      }
}
```

0: a new record, new level lists,
1: new level 1 list, new level 2 list
2: new element in level 2 list

Repetition level	Value
0	a
2	b
2	c
1	d
2	e
2	f
2	g
0	h
1	i
2	j


0
1
2
R
Nested lists
level1
level2: a
0
new record
level2: b
2
new level2 entry
level2: c
2
new level2 entry
level1
level2: d
1
new level1 entry
level2: e
2
new level2 entry
level2: f
2
new level2 entry
level2: g
2
new level2 entry
Nested lists
level1
level2: h
0
new record
level1
level2: i
1
new level1 entry
level2: j
2
new level2 entry

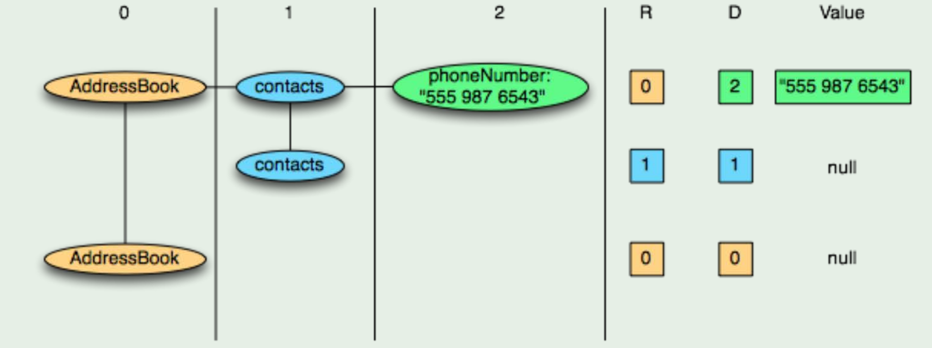
Striping and assembly

```
message AddressBook {
    required string owner;
    repeated string ownerPhoneNumbers;
    repeated group contacts {
        required string name;
        optional string phoneNumber;
    }
}
```

considering only `contacts.phoneNumber`:

```
AddressBook {
    contacts: {
        phoneNumber: "555 987 6543"
    }
    contacts: {
    }
}
AddressBook {
}
```

Striping



Striped representation

contacts.phoneNumber		
R	D	Value
0	2	"555 987 6543"
1	1	NULL
0	0	NULL

Striped representation

contacts.phoneNumber		
R	D	Value
0	2	"555 987 6543"
1	1	NULL
0	0	NULL

Assembly

```
R=0, D=2, Value = "555 987 6543":
        R = 0 means a new record.
            We recreate the nested records from the root
            until the definition level (here 2)
        D = 2 which is the maximum.
            The value is defined and is inserted.
R=1, D=1:
        R = 1 means a new entry in the contacts list at level 1.
        D = 1 means contacts is defined but not phoneNumber,
            so we just create an empty contacts.
R=0, D=0:
        R = 0 means a new record. we create the nested records
            from the root until the definition level
        D = 0 => contacts is actually null, so we only
            have an empty AddressBook
```

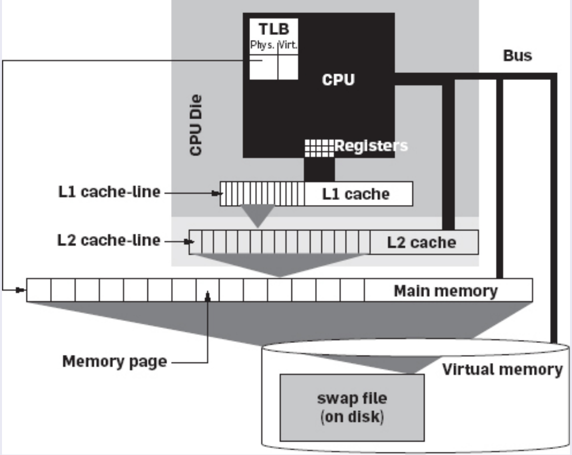
MonetDB[2]

In the past decades, advances in speed of commodity CPUs have far outpaced advances in RAM latency. Main-memory access has become a performance bottleneck for many computer applications: "memory wall."

- Redesign of the query execution model to better exploit pipelined CPU architectures and CPU instruction caches;
- use of columnar rather than row-wise data storage to better exploit CPU data caches;
- design of new cache-conscious query processing algorithms;
- design and automatic calibration of memory cost models to choose and tune these cache-conscious algorithms in the query optimizer.

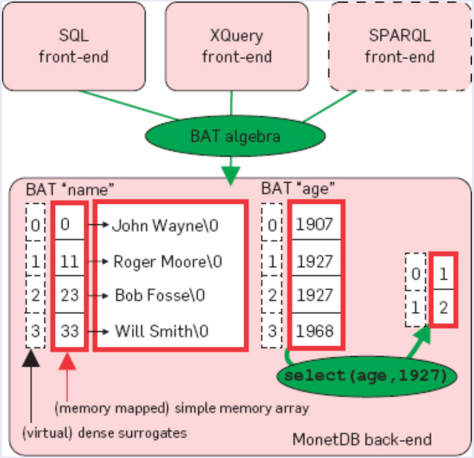
[2]Breaking the Memory Wall in MonetDB. Peter A. Boncz, Martin L. Kersten, Stefan Manegold. Communications of the ACM, Vol. 51 No. 12

CPU-Architecture



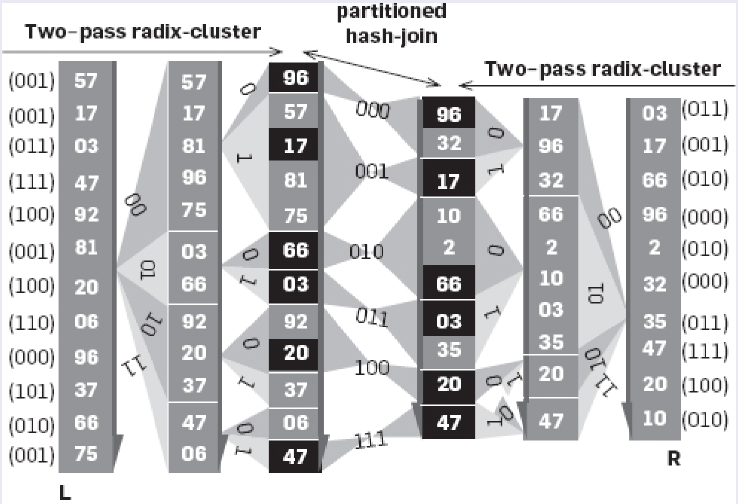
A translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache.

MonetDB Architecture



BAT: Binary Asssociation Table

Cash-conscious join: Partitioned hash-join



Black tuples hit (lowest 3-bits of values in parenthesis)

- Divide a relation U into H clusters using P sequential passes.
- Radix-clustering is on the lower B bits of the integer hash-value of a column. Each pass clusters tuples on B_p bits, starting with the leftmost bits, where $\sum_{p=1}^{P} B_p = B$.
- The number of clusters created by the Radix-Cluster is $H = \prod_{p=1}^{P} H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones.
- When the algorithm starts, the entire relation is considered one single cluster, and is subdivided into $H_1 = 2^{B_1}$ clusters. The next pass takes these clusters and subdivides each into $H_1 = 2^{B_1}$ new ones, yielding $H_1 * H_2$ clusters in total, etc.

- Divide a relation U into H clusters using P sequential passes.
- Radix-clustering is on the lower B bits of the integer hash-value of a column. Each pass clusters tuples on B_p bits, starting with the leftmost bits, where $\sum_{p=1}^{P} B_p = B$.
- The number of clusters created by the Radix-Cluster is $H = \prod_{p=1}^{P} H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones.
- When the algorithm starts, the entire relation is considered one single cluster, and is subdivided into $H_1 = 2^{B_1}$ clusters. The next pass takes these clusters and subdivides each into $H_1 = 2^{B_1}$ new ones, yielding $H_1 * H_2$ clusters in total, etc.

- Keep $H_x = 2^{B_x}$ smaller than the number of cache lines and the number of TLB entries, this will completely avoid both TLB and cache thrashing.
- After Radix-Clustering a column on B bits, all tuples that have the same B lowest bits in its column hash-value, appear consecutively in the relation, typically forming clusters of $\mid U \mid /2^B$ tuples.

- Divide a relation U into H clusters using P sequential passes.
- Radix-clustering is on the lower B bits of the integer hash-value of a column. Each pass clusters tuples on B_p bits, starting with the leftmost bits, where $\sum_{p=1}^{P} B_p = B$.
- The number of clusters created by the Radix-Cluster is $H = \prod_{p=1}^{P} H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones.
- When the algorithm starts, the entire relation is considered one single cluster, and is subdivided into $H_1 = 2^{B_1}$ clusters. The next pass takes these clusters and subdivides each into $H_1 = 2^{B_1}$ new ones, yielding $H_1 * H_2$ clusters in total, etc.

- Keep $H_x = 2^{B_x}$ smaller than the number of cache lines and the number of TLB entries, this will completely avoid both TLB and cache thrashing.
- After Radix-Clustering a column on B bits, all tuples that have the same B lowest bits in its column hash-value, appear consecutively in the relation, typically forming clusters of $\mid U \mid /2^B$ tuples.

2.2 Not only SQL (NoSQL) Database System

Not "No SQL", but "Not only SQL"

Non-relational databases are the answer to new data management challenges:

- *Complex structures*: e.g. graphs,
- *Schema independence*: e.g. to support flexible integration of data from different sources into the same store,
- *Sparseness*: e.g. do not store NULL-values,
- *Self-descriptiveness*: e.g. attach meta-data to individual values,
- *Variability*: e.g. constantly changing data and schemata,
- *Scalability*: data is processed in a distributed fashion,
- *Volume*: large volumns have to be processsed.

A NoSQL database system may be non-relational, support non-standard query languages, support schema evolution and independance, support data distribution with possibly a weaker form of consistency.

2.2 Not only SQL (NoSQL) Database System

Not "No SQL", but "Not only SQL"

Non-relational databases are the answer to new data management challenges:

- *Complex structures*: e.g. graphs,
- *Schema independence*: e.g. to support flexible integration of data from different sources into the same store,
- *Sparseness*: e.g. do not store NULL-values,
- *Self-descriptiveness*: e.g. attach meta-data to individual values,
- *Variability*: e.g. constantly changing data and schemata,
- *Scalability*: data is processed in a distributed fashion,
- *Volume*: large volumns have to be processsed.

A NoSQL database system may be non-relational, support non-standard query languages, support schema evolution and independance, support data distribution with possibly a weaker form of consistency.

Key-Value Store

Key-value pair: $\langle key, value \rangle$.

- *key*: string, unique identifier,
- *value*: string.

Prototype of a schema-less database: arbitrary key-value pairs can be stored and retrieved; *simple, but quick*.

- Disadvantage: values cannot be searched, no query language;
- Advantage: data can be easily distributed.

Key-Value Store

Key-value pair: $\langle key, value \rangle$.

- *key*: string, unique identifier,
- *value*: string.

Prototype of a schema-less database: arbitrary key-value pairs can be stored and retrieved; *simple, but quick*.

- Disadvantage: values cannot be searched, no query language;
- Advantage: data can be easily distributed.

Values may have a type other than *string*, e.g. collection types *list* or *array*, even nested.

$\Longrightarrow$ *Document databases*; data formats XML or JSON: a key-value pair may be the value of another key-value pair.

Document database systems, e.g. MongoDB, CouchDB provide additional processing functionality.

Key-Value Store

Key-value pair: $\langle key, value \rangle$.

- *key*: string, unique identifier,
- *value*: string.

Prototype of a schema-less database: arbitrary key-value pairs can be stored and retrieved; *simple, but quick*.

- Disadvantage: values cannot be searched, no query language;
- Advantage: data can be easily distributed.

Values may have a type other than *string*, e.g. collection types *list* or *array*, even nested.

$\Longrightarrow$ *Document databases*; data formats XML or JSON: a key-value pair may be the value of another key-value pair.

Document database systems, e.g. MongoDB, CouchDB provide additional processing functionality.

Wide-column Store

Example Google BigTable/Apache HBase: extensible record stores

Data is stored in tables, where records have the ability to hold very large numbers of dynamic columns. Tables are typically not normalized!

- Column management implements the concept of *column families*.
- Akin to key-value stores, *multidimensional* keys are mapped to a value.
- The column names as well as the record keys are not fixed; a record can have billions (?) of columns.
- Characteristic of being schema-free is shared with document stores, however the implementation is very different.
- Wide column stores must not be confused with the column oriented storage in some relational systems.

Wide-column Store

Example Google BigTable/Apache HBase: extensible record stores

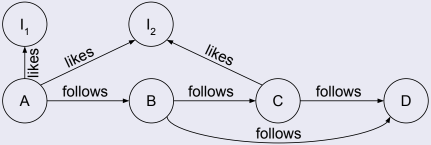
Data is stored in tables, where records have the ability to hold very large numbers of dynamic columns. Tables are typically not normalized!

- Column management implements the concept of *column families*.
- Akin to key-value stores, *multidimensional* keys are mapped to a value.
- The column names as well as the record keys are not fixed; a record can have billions (?) of columns.
- Characteristic of being schema-free is shared with document stores, however the implementation is very different.
- Wide column stores must not be confused with the column oriented storage in some relational systems.

2.3 Graph-DB
(A) Edge-labelled Graphs: RDF

To draw an RDF graph, for each Subject-Predicate-Object-triple introduce $s \xrightarrow{p} o$.

Subject	Predicate	Object
A	likes	I_1
A	likes	I_2
C	likes	I_2
A	follows	B
B	follows	C
B	follows	D
C	follows	D



RDF-storage by Relational Databases.

Different approaches:

- Giant Tripel-Table
- Vertical Partitioning
- Property-Table

Other options: Native RDF-stores, Graph-Databases.

Triple Table

Triple-Table

Subj.	Prop.	Obj.
ID1	type	BookType
ID1	title	“XYZ”
ID1	author	“Fox, Joe”
ID1	copyright	“2001”
ID2	type	CDType
ID2	title	“ABC”
ID2	artist	“Orr, Tim”
ID2	copyright	“1985”
ID2	language	“French”
ID3	type	BookType
ID3	title	“MNO”
ID3	language	“English”
ID4	type	DVDType
ID4	title	“DEF”
ID5	type	CDType
ID5	title	“GHI”
ID5	copyright	“1995”
ID6	type	BookType
ID6	copyright	“2004”

Property Table

Property Table Approach

Property Table

Subj.	Type	Title	copyright
ID1	BookType	"XYZ"	"2001"
ID2	CDType	"ABC"	"1985"
ID3	BookType	"MNP"	NULL
ID4	DVDType	"DEF"	NULL
ID5	CDType	"GHI"	"1995"
ID6	BookType	NULL	"2004"

Left-Over Triples

Subj.	Prop.	Obj.
ID1	author	"Fox, Joe"
ID2	artist	"Orr, Tim"
ID2	language	"French"
ID3	language	"English"

Clustered property table

Class: BookType

Subj.	Title	Author	copyright
ID1	"XYZ"	"Fox, Joe"	"2001"
ID3	"MNP"	NULL	NULL
ID6	NULL	NULL	"2004"

Class: CDType

Subj.	Title	Artist	copyright
ID2	"ABC"	"Orr, Tim"	"1985"
ID5	"GHI"	NULL	"1995"

Left-Over Triples

Subj.	Prop.	Obj.
ID2	language	"French"
ID3	language	"English"
ID4	type	DVDType
ID4	title	"DEF"

Property-class table

Discussion: Triple vs. Property Tables

- Triple table:

  Advantage: Simplicity.

  Problems: Multi-way self-joins may result, which, when not selective, are very expensive.

- Property table:

  Advantage: Subject-Subject self-joins are reduced; attribute-typing possible.

  Problems:

  - Queries with unspecified property values are problematic.
  - NULL-values may yield sparse tables.
  - What to do with multi-valued attributes?

Discussion: Triple vs. Property Tables

- Triple table:

  Advantage: Simplicity.

  Problems: Multi-way self-joins may result, which, when not selective, are very expensive.

- Property table:

  Advantage: Subject-Subject self-joins are reduced; attribute-typing possible.

  Problems:

  - Queries with unspecified property values are problematic.
  - NULL-values may yield sparse tables.
  - What to do with multi-valued attributes?

Vertically partitioned approach

Table per property

Type

ID1	BookType
ID2	CDType
ID3	BookType
ID4	DVDType
ID5	CDType
ID6	BookType

Author

ID1	"Fox, Joe"

Title

ID1	"XYZ"
ID2	"ABC"
ID3	"MNO"
ID4	"DEF"
ID5	"GHI"

Artist

ID2	"Orr, Tim"

Copyright

ID1	"2001"
ID2	"1985"
ID5	"1995"
ID6	"2004"

Language

ID2	"French"
ID3	"English"

Author

ID1	"Fox, Joe"
ID1	"Green, John"

Multi-valued attributes.

Vertically partitioned approach

Each table is sorted by subject. Object column can optionally be indexed.

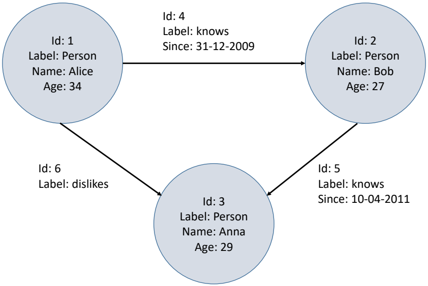
- Advantages:
  - Support for multi-valued attributes.
  - Support for heterogenous records.
  - Reduced I/O-costs: only that what indeed is needed will be accessed.
  - No property-clustering required.
- Problems:
  - Increased number of joins.
  - Queries with unspecified properties.
  - Inserts.

Vertically partitioned approach

Each table is sorted by subject. Object column can optionally be indexed.

- Advantages:
  - Support for multi-valued attributes.
  - Support for heterogenous records.
  - Reduced I/O-costs: only that what indeed is needed will be accessed.
  - No property-clustering required.
- Problems:
  - Increased number of joins.
  - Queries with unspecified properties.
  - Inserts.

(B) Property Graphs



Storing in relational tables

Nodes	NodeID	NodeLabel
	1	Person
	2	Person
	3	Person

PersonAttributes	NodeID	Name	Age
	1	Alice	34
	2	Bob	27
	3	Charlene	29

Edges	EdgeID	EdgeLabel	Source	Target
	4	knows	1	2
	5	knows	2	3
	6	dislikes	1	3

KnowsAttributes	EdgeID	Since
	4	31-12-2009
	5	10-04-2011

Native Graph Storage

- Property stores: key-value pairs;
- Graph structure: by node and relationship stores, sophisticated highly efficient linked list structures.

  Akin adjacency lists.