Processing of Very Large Data

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Software Development Technologies
Master studies, first semester
Academic year 2017/18 (winter course)

Review of the Previous Lecture

- Mining of massive datasets.
- Evolution of database systems.
- Dimensional modeling.
- ETL and OLAP systems.
- MapReduce in Spark

Processing of very large data

- Physical data organization: row-based, column-based, key-values stores, multi-dimensional arrays, etc.
- Partitioning and sharding (Map-Reduce, distributed databases).
- Data access: hashing and sorting ($\rightarrow$ tree-based indexing).
- Advanced data structures: multi-dimensional indexes, inverted lists, bitmaps, special-purpose indexes.
- Summarization, materialization, and denormalization.
- Data compression.
- Approximate query processing.
- Probabilistic data structures and algorithms.
- Data schemas: star schema, flexible schemas.

Outline

1. Physical Storage
2. Denormalization and Summarization
3. Data Access
4. Data Partitioning
5. Summary

Outline

Physical storage

- Row-based,
- Column-based,
- Key-values stores,
- Multi-dimensional arrays,
- Dense vs. sparse structures.

Physical storage

- The following table can be stored in different ways:

Year	Products	Sales
2010	Mountain	5076
2010	Road	4005
2010	Touring	3560
2011	Mountain	6503
2011	Road	4503
2011	Touring	3445

Physical storage

- Row-based storage:

  001: 2010, Mountain, 5076, **002**: 2010, Road, 4005, **003**: 2010, Touring, 3560, **004**: 2011, Mountain, 6503, **005**: 2011, Road, 4503 **006**: 2011, Touring, 3445.

Physical storage

- Row-based storage:

  001: 2010, Mountain, 5076, **002**: 2010, Road, 4005, **003**: 2010, Touring, 3560, **004**: 2011, Mountain, 6503, **005**: 2011, Road, 4503 **006**: 2011, Touring, 3445.

- Column-based storage:

  Y: 2010, 2010, 2010, 2011, 2011, 2011, **P**: Mountain, Road, Touring, Mountain, Road, Touring, **S**: 5076, 5004, 3560, 6503, 4503, 3445.

  or

  Y: 2010: **001**, **002**, **003**, 2011: **004**, **005**, **006**, **P**: Mountain: **001**, **004**, Road: **002**, **005**, Touring: **003**, **006**, **S**: 5076: **001**, 4005, **002**, 3560: **003**, 6503: **004**, 4503: **005**, 3445: **006**

Physical storage

- Key-value pairs:

  001,Y: 2010, **002,Y**: 2010, **003,Y**: 2010, **004,Y**: 2011, **005,Y**: 2011, **006,Y**: 2011, **001,P**: Mountain, **002,P**: Road, **003,P**: Touring, **004,P**: Mountain, **005,P**: Road, **006,P**: Touring, **001,S**: 5076, **002,S**: 4005, **003,S**: 3506, **004,S**: 6503, **005,S**: 4503, **006,S**: 3445

Physical storage

- Key-value pairs:

  001,Y: 2010, **002,Y**: 2010, **003,Y**: 2010, **004,Y**: 2011, **005,Y**: 2011, **006,Y**: 2011, **001,P**: Mountain, **002,P**: Road, **003,P**: Touring, **004,P**: Mountain, **005,P**: Road, **006,P**: Touring, **001,S**: 5076, **002,S**: 4005, **003,S**: 3506, **004,S**: 6503, **005,S**: 4503, **006,S**: 3445
- Multidimensional array:

  Y: 2010, 2011, **P**: Mountain, Road, Touring, **S**: 5076, 4005, 3560, 6503, 4503, 3445

Outline

1. Physical Storage
2. Denormalization and Summarization
3. Data Access
4. Data Partitioning
5. Summary

Denormalization and summarization

- Relational and multidimensional model with summarizations:

Year	Products	Sales
2010	Mountain	5076
2010	Road	4005
2010	Touring	3560
2011	Mountain	6503
2011	Road	4503
2011	Touring	3445
2010	*	12461
2011	*	14451
*	Mountain	11579
*	Road	6503
*	Touring	7005
*	*	27092

	Product	Mountain	Road	Touring	All
Year	2010	5076	4005	3560	12641
	2011	6503	4503	3445	14451
	All	11579	8508	7005	27092

Denormalization and summarization

- Trade-off between query performance and load performance
- To improve performance of query processing:
  - Precompute as much as possible,
  - Build additional data structures like indexes.
- The costs of the above are:
  - Disk space,
  - Load time,
  - Processing time of building and updating of data structures.

Denormalization and summarization

- Typical techniques:

Denormalization and summarization

- Typical techniques:
  - Aggregate (summary) tables: aggregating fact tables across some dimensions,

Denormalization and summarization

- Typical techniques:
  - Aggregate (summary) tables: aggregating fact tables across some dimensions,
  - Dimension aggregates: for example, base date dimension, monthly aggregate dimension, yearly aggregate dimension,

Denormalization and summarization

- Typical techniques:
  - Aggregate (summary) tables: aggregating fact tables across some dimensions,
  - Dimension aggregates: for example, base date dimension, monthly aggregate dimension, yearly aggregate dimension,
  - ROLAP: Materialized views or indexed views,

Denormalization and summarization

- Typical techniques:
  - Aggregate (summary) tables: aggregating fact tables across some dimensions,
  - Dimension aggregates: for example, base date dimension, monthly aggregate dimension, yearly aggregate dimension,
  - ROLAP: Materialized views or indexed views,
  - MOLAP: Subcubes or aggregations.

Denormalization and summarization

- Store in data warehouse results useful for common queries.

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:
  - every,

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:
  - every,
  - none,

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:
  - every,
  - none,
  - some.

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:
  - every,
  - none,
  - some.
- The problem relies in selection of cuboids to be materialized (size, sharing, access frequency):

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:
  - every,
  - none,
  - some.
- The problem relies in selection of cuboids to be materialized (size, sharing, access frequency):
  - high number of materialized cuboids $\rightarrow$ huge size of data warehouse.

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:
  - every,
  - none,
  - some.
- The problem relies in selection of cuboids to be materialized (size, sharing, access frequency):
  - high number of materialized cuboids $\rightarrow$ huge size of data warehouse.
  - small number of materialized cuboids $\rightarrow$ slow query processing.

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:
  - every,
  - none,
  - some.
- The problem relies in selection of cuboids to be materialized (size, sharing, access frequency):
  - high number of materialized cuboids $\rightarrow$ huge size of data warehouse.
  - small number of materialized cuboids $\rightarrow$ slow query processing.
- Aggregates should be computed from previously computed aggregates, rather than from the base fact table.

Denormalization and summarization

- Store in data warehouse results useful for common queries.
- Three strategies to materialize cuboids:
  - every,
  - none,
  - some.
- The problem relies in selection of cuboids to be materialized (size, sharing, access frequency):
  - high number of materialized cuboids $\rightarrow$ huge size of data warehouse.
  - small number of materialized cuboids $\rightarrow$ slow query processing.
- Aggregates should be computed from previously computed aggregates, rather than from the base fact table.
- The problem appears with maintenance of the materialized views: recomputation and incremental updating.

View vs. materialized views

- **View** is a derived relation defined in terms of base (stored) relations.
- **Materialized view** (or indexed view) is a view stored in a database that is updated from the original base tables from time to time.

Query re-write

- **Query rewrite**: transforms a given query expressed in terms of base tables or views into a statement accessing one or more materialized views (e.g., aggregates) that are defined on the detail tables.
- The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the query.

Query re-write

- **Example**: Materialized views in SQL
  - Materialized view V:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product id = p.id AND p.year_of_release > 1990
GROUP BY p.name, p.year_of_release;
```

  - Materialized view V consists of:
    - Join of the fact table with dimension table,
    - Group by dimension attributes,
    - Aggregation of measures included in fact table.

Query re-write

- **Example**: Materialized views in SQL

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product_id = p.id AND p.year_of_release > 1991
GROUP BY p.name, p.year_of_release;
```

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product_id = p.id AND p.year_of_release > 1991
GROUP BY p.name, p.year_of_release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year_of_release > 1991;
```

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

    ```
    SELECT p.name, p.year_of_release, sum(s.price) as price
    FROM Sales s, Product p
    WHERE s.product_id = p.id AND p.year_of_release > 1991
    GROUP BY p.name, p.year_of_release;
    ```
  - Query rewrite

    ```
    SELECT p.name, p.year_of_release, price
    FROM V
    WHERE year_of_release > 1991;
    ```
  - The query re-write is possible since the exact match holds:

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product_id = p.id AND p.year_of_release > 1991
GROUP BY p.name, p.year_of_release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year_of_release > 1991;
```

  - The query re-write is possible since the exact match holds:
    - all the projected columns are also in V,

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

    ```
    SELECT p.name, p.year_of_release, sum(s.price) as price
    FROM Sales s, Product p
    WHERE s.product_id = p.id AND p.year_of_release > 1991
    GROUP BY p.name, p.year_of_release;
    ```

  - Query rewrite

    ```
    SELECT p.name, p.year_of_release, price
    FROM V
    WHERE year_of_release > 1991;
    ```

  - The query re-write is possible since the exact match holds:
    - all the projected columns are also in V,
    - the same aggregate functions are used on all measures,

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product_id = p.id AND p.year_of_release > 1991
GROUP BY p.name, p.year_of_release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year_of_release > 1991;
```

  - The query re-write is possible since the exact match holds:
    - all the projected columns are also in V,
    - the same aggregate functions are used on all measures,
    - all selection conditions in the query imply the selection conditions in V,

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, p.year_of_release, sum(s.price) as price
FROM Sales s, Product p
WHERE s.product_id = p.id AND p.year_of_release > 1991
GROUP BY p.name, p.year_of_release;
```

  - Query rewrite

```
SELECT p.name, p.year_of_release, price
FROM V
WHERE year_of_release > 1991;
```

  - The query re-write is possible since the exact match holds:
    - all the projected columns are also in V,
    - the same aggregate functions are used on all measures,
    - all selection conditions in the query imply the selection conditions in V,
    - the attributes present in selection conditions that are strictly stronger than selection conditions defined in V, are also present in V.

Query re-write

- **Example**: Materialized views in SQL

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, sum(s.price) FROM
Sales s, Product p
WHERE s.product_id = p.id AND p.year_of_release > 1995
GROUP BY p.name;
```

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, sum(s.price) FROM
Sales s, Product p
WHERE s.product_id = p.id AND p.year_of_release > 1995
GROUP BY p.name;
```

  - Query rewrite

```
SELECT name,sum(price)
FROM V
WHERE year_of_release > 1995
GROUP BY name;
```

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

```
SELECT p.name, sum(s.price) FROM
Sales s, Product p
WHERE s.product_id = p.id AND p.year_of_release > 1995
GROUP BY p.name;
```

  - Query rewrite

```
SELECT name,sum(price)
FROM V
WHERE year_of_release > 1995
GROUP BY name;
```

  - The query re-write is possible since the additional grouping can be performed on V (non-exact match):

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

    ```
    SELECT p.name, sum(s.price) FROM
    Sales s, Product p
    WHERE s.product_id = p.id AND p.year_of_release > 1995
    GROUP BY p.name;
    ```
  - Query rewrite

    ```
    SELECT name,sum(price)
    FROM V
    WHERE year_of_release > 1995
    GROUP BY name;
    ```
  - The query re-write is possible since the additional grouping can be performed on V (non-exact match):
    - all attributes involved in query are present in V,

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:
    ```
    SELECT p.name, sum(s.price) FROM
    Sales s, Product p
    WHERE s.product_id = p.id AND p.year_of_release > 1995
    GROUP BY p.name;
    ```
  - Query rewrite
    ```
    SELECT name,sum(price)
    FROM V
    WHERE year_of_release > 1995
    GROUP BY name;
    ```
  - The query re-write is possible since the additional grouping can be performed on V (non-exact match):
    - all attributes involved in query are present in V,
    - selection conditions are stronger,

Query re-write

- **Example**: Materialized views in SQL
  - Exemplary query:

    ```
    SELECT p.name, sum(s.price) FROM
    Sales s, Product p
    WHERE s.product_id = p.id AND p.year_of_release > 1995
    GROUP BY p.name;
    ```
  - Query rewrite

    ```
    SELECT name,sum(price)
    FROM V
    WHERE year_of_release > 1995
    GROUP BY name;
    ```
  - The query re-write is possible since the additional grouping can be performed on V (non-exact match):
    - all attributes involved in query are present in V,
    - selection conditions are stronger,
    - grouping is more general.

Maintenance of materialized views

- Let V be the materialized view defined by a query Q over a set R of relations

$$V = Q(R)\,.$$

Maintenance of materialized views

- Let V be the materialized view defined by a query Q over a set R of relations
$$V = Q(R)\,.$$
- When the relations in R are updated, then V becomes inconsistent.

Maintenance of materialized views

- Let V be the materialized view defined by a query Q over a set R of relations
$$V = Q(R)\,.$$
- When the relations in R are updated, then V becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between R and V.

Maintenance of materialized views

- Let V be the materialized view defined by a query Q over a set R of relations
$$V = Q(R)\,.$$
- When the relations in R are updated, then V becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between R and V.
- Different aspects:

Maintenance of materialized views

- Let V be the materialized view defined by a query Q over a set R of relations
$$V = Q(R)\,.$$
- When the relations in R are updated, then V becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between R and V.
- Different aspects:
  - Immediate and delayed refresh.

Maintenance of materialized views

- Let V be the materialized view defined by a query Q over a set R of relations
$$V = Q(R)\,.$$
- When the relations in R are updated, then V becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between R and V.
- Different aspects:
  - Immediate and delayed refresh.
  - Full refresh and view maintenance.

Maintenance of materialized views

- Let V be the materialized view defined by a query Q over a set R of relations
$$V = Q(R)\,.$$
- When the relations in R are updated, then V becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between R and V.
- Different aspects:
  - Immediate and delayed refresh.
  - Full refresh and view maintenance.
  - Maintainable and partially maintainable views.

Maintenance of materialized views

- Let V be the materialized view defined by a query Q over a set R of relations
$$V = Q(R)\,.$$
- When the relations in R are updated, then V becomes inconsistent.
- View refreshment is the process that reestablishes the consistency between R and V.
- Different aspects:
    - Immediate and delayed refresh.
    - Full refresh and view maintenance.
    - Maintainable and partially maintainable views.
- **Example**: How to maintain the materialized view defined below?

```
V = SELECT min(A.a) FROM A
```

Outline

Data access

- Hashing
- Sorting ($\rightarrow$ tree-based indexing).

Grouping

- **Group-by** is usually performed in the following way:

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,
    - Sort or re-hash within each bucket to resolve collisions.

Grouping

- **Group-by** is usually performed in the following way:
  - Partition tuples on grouping attributes: tuples in same group are placed together, and in different groups separated,
  - Scan tuples in each partition and compute aggregate expressions.
- Two techniques for partitioning:
  - Sorting
    - Sort by the grouping attributes,
    - All tuples with same grouping attributes will appear together in sorted list.
  - Hashing
    - Hash by the grouping attributes,
    - All tuples with same grouping attributes will hash to same bucket,
    - Sort or re-hash within each bucket to resolve collisions.
- In OLAP queries use intermediate results to compute more general group-bys.

Grouping

- **Example**: Grouping by sorting (Month, City):

Month	City	Sale
March	Poznań	105
March	Warszawa	135
March	Poznań	50
May	Warszawa	100
April	Poznań	150
April	Kraków	175
May	Poznań	70
May	Warszawa	75

Grouping

- **Example**: Grouping by sorting (Month, City):

Month	City	Sale
March	Poznań	105
March	Warszawa	135
March	Poznań	50
May	Warszawa	100
April	Poznań	150
April	Kraków	175
May	Poznań	70
May	Warszawa	75

$\longrightarrow$

Month	City	Sale
March	Poznań	105
March	Poznań	50
March	Warszawa	135
April	Poznań	150
April	Kraków	175
May	Poznań	70
May	Warszawa	75
May	Warszawa	100

Grouping

- **Example**: Grouping by sorting (Month, City):

Month	City	Sale
March	Poznań	105
March	Warszawa	135
March	Poznań	50
May	Warszawa	100
April	Poznań	150
April	Kraków	175
May	Poznań	70
May	Warszawa	75

$\longrightarrow$

Month	City	Sale
March	Poznań	105
March	Poznań	50
March	Warszawa	135
April	Poznań	150
April	Kraków	175
May	Poznań	70
May	Warszawa	75
May	Warszawa	100

$\downarrow$

Month	City	Sale
March	Poznań	155
March	Warszawa	135
April	Poznań	150
April	Kraków	175
May	Poznań	70
May	Warszawa	175

Aggregates computed from aggregates

All rows and columns

Academic_year	Name	AVG(Grade)
2011/2	Stefanowski	4.2
2011/2	Słowiński	4.1
2012/3	Stefanowski	4.0
2012/3	Słowiński	3.8
2013/4	Stefanowski	3.9
2013/4	Słowiński	3.6
2013/4	Dembczyński	4.8

Academic_year	AVG(Grade)
2011/2	4.15
2012/3	3.85
2013/4	3.8

Name	AVG(Grade)
Stefanowski	3.9
Słowiński	3.6
Dembczyński	4.8

AVG(Grade)
3.95

Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.

Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.
- An index is a “thin” copy of a relation (not all columns from the relation are included, the index is sorted in a particular way).

Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.
- An index is a “thin” copy of a relation (not all columns from the relation are included, the index is sorted in a particular way).
- Index-only plans use small indexes in place of large relations.

Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.
- An index is a “thin” copy of a relation (not all columns from the relation are included, the index is sorted in a particular way).
- Index-only plans use small indexes in place of large relations.
- Query processing on indexes – without accessing base tables.

Indexes

- Indexes allow efficient search on some attributes due to the way they are organized.
- An index is a "thin" copy of a relation (not all columns from the relation are included, the index is sorted in a particular way).
- Index-only plans use small indexes in place of large relations.
- Query processing on indexes – without accessing base tables.
- Indexes on two and more columns.

Indexes

- B-Trees,
- Inverted lists,
- Bitmap index,
- Bit-sliced index,
- Projection index,
- Join index.

Bitmap index

- Bitmap indexes use bit arrays (commonly called "bitmaps") to encode values on a given attribute and answer queries by performing bitwise logical operations on these bitmaps.

Bitmap index

- Bitmap indexes use bit arrays (commonly called "bitmaps") to encode values on a given attribute and answer queries by performing bitwise logical operations on these bitmaps.

Customer	City	Car
C1	Detroit	Ford
C2	Chicago	Honda
C3	Detroit	Honda
C4	Poznań	Ford
C5	Paris	BMW
C6	Paris	Nissan

Bitmap index

- Bitmap indexes use bit arrays (commonly called "bitmaps") to encode values on a given attribute and answer queries by performing bitwise logical operations on these bitmaps.

Customer	City	Car
C1	Detroit	Ford
C2	Chicago	Honda
C3	Detroit	Honda
C4	Poznań	Ford
C5	Paris	BMW
C6	Paris	Nissan

$\downarrow$

Customer	Chicago	Detroit	Paris	Poznań
C1	0	1	0	0
C2	1	0	0	0
C3	0	1	0	0
C4	0	0	0	1
C5	0	0	1	0
C6	0	0	1	0

$\rightarrow$

Bitmap	Array of bytes
Chicago	010000 (00)
Detroit	101000 (00)
Paris	010011 (00)
Poznań	000100 (00)

Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),

Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,

Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,
- Usually bitmap indexes are compressed,

Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,
- Usually bitmap indexes are compressed,
- Works poorly for high cardinality domains since the number of bitmaps increases,

Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,
- Usually bitmap indexes are compressed,
- Works poorly for high cardinality domains since the number of bitmaps increases,
- Difficult to maintain – need reorganization when relation sizes change (new bitmaps)

Bitmap index

- Allows the use of efficient bit operations to answer some queries (hardware support for bitmap operations),
- Very efficient for certain types of queries: selection on two attributes,
- Usually bitmap indexes are compressed,
- Works poorly for high cardinality domains since the number of bitmaps increases,
- Difficult to maintain – need reorganization when relation sizes change (new bitmaps)
- Can be used with B-Trees.

Compressing Bitmaps

- Compression Pros and Cons
  - Reduce storage space $\rightarrow$ reduce number of I/Os required
  - Need to compress/uncompress $\rightarrow$ increase CPU work required
  - Operate directly on compressed bitmap $\rightarrow$ improved performance
- Bitmaps consist mostly of zeros
- Compression via **run length encoding**:
  - Example: 00000001000010000000000001100000
  - Just record the length of sequences composed of zeros or ones:
  - Store this as "7,1,4,1,12,2,5",
  - alternatively: record the number of zeros between adjacent ones
  - Store this as "7,4,12,0,5".

Compressing Bitmaps

- Simple run length encoding is not sufficient and we need structured encoding:
  - Example: 00000001000010000000000001100000
  - We can store this as "7,4,12,0,5"
  - But we cannot use a bitmap to encode the above since:
  - 11110011000101 could be read not only as 7,4,12,0,5: (111)(100)(1100)(0)(101),
  - but also as 3,25,8,2,1: (11)(11001)(1000)(10)(1).

γ codes

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101$
- Length is the length of offset.
- For 13 (offset 101), this is 3.
- Encode length in unary code: 1110.
- Gamma code of 13 is the concatenation of length and offset: 1110101.

Unary code

- Represent n as n 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is 11111111111111111111111111111111111111110
  .

Gamma code examples

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	1111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110,0000000001

Length of gamma code

- The length of **offset** is $\lfloor \log_2 G \rfloor$ bits.
- The length of **length** is $\lfloor \log_2 G \rfloor + 1$ bits,
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
- γ codes are always of odd length.
- Gamma codes are within a factor of 2 of the optimal encoding length $\log_2 G$.
  - Assuming equal-probability gaps – but the distribution is actually highly skewed.
  - We can use gamma codes for any distribution.
  - The code is universal.

Bitmap compression with BBC (Byte-Aligned Bitmap Code) codes

- Divide bitmap into bytes:
  - **Gap** bytes are all zeros
  - **Tail** bytes contain some ones
  - A **chunk** consists of some gap bytes followed by some tail bytes
- Encode chunks:
  - Header byte
  - Gap length bytes (sometimes)
  - Verbatim tail bytes (sometimes)

Exemplary bitmap:

00000000 00000000 00010000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 01000000 00100010

Bitmap compression with BBC codes

- Number of gap bytes:
  - 0-6: Gap length stored in header byte
  - 7-127: One gap-length byte follows header byte
  - 128-32767: Two gap-length bytes follow header byte
- "Special" tail:
  - Tail consists of only 1 byte
  - The tail byte has only 1 non-zero bit
  - Non-special tails are stored verbatim (uncompressed)
- Number of tail bytes:
  - Number of tail bytes is stored in header byte
  - Special tails are encoded by indicating which bit is set

Bitmap compression with BBC codes

- Header byte:
  - Bits 1-3: length of (short) gap
    - Gaps of length 0-6 do not require gap length bytes
    - 111 = gap length > 6
  - Bit 4: Is the tail special?
  - Bits 5-8:
    - Number of verbatim bytes (if bit 4=0)
    - Index of non-zero bit in tail byte (if bit 4 = 1)

Bitmap compression with BBC codes

- Gap length bytes:
  - Either one or two bytes
  - Only present if bits 1-3 of header are 111
  - Gap lengths of 7-127 encoded in single byte
  - Gap lengths of 128-32767 encoded in 2 bytes
  - 1st bit of 1st byte set to 1 to indicate 2-byte case
- Verbatim bytes:
  - 0-15 uncompressed tail bytes
  - Number is indicated in header

Bitmap compression with BBC codes

Exemplary bitmap:

00000000 00000000 00010000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 01000000 00100010

Bitmap compression with BBC codes

Exemplary bitmap:

00000000 00000000 00010000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 01000000 00100010

Bitmap after compression

01010100 11100010 00001101 01000000 00100010

Bitmap compression with BBC codes

Exemplary bitmap:

00000000 00000000 00010000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 01000000 00100010

- Bitmap consists of two chunks:
  - Chunk 1
    - Bytes 1-3
    - Two gap bytes, one tail byte
    - Encoding: (010)(1)(0100)
    - No gap length bytes since gap length < 7
    - No verbatim bytes since tail is special

Bitmap after compression

01010100 11100010 00001101 01000000 00100010

Bitmap compression with BBC codes

Exemplary bitmap:

00000000 00000000 00010000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 01000000 00100010

- Bitmap consists of two chunks:
  - Chunk 2
    - Bytes 4-18
    - 13 gap bytes, two tail bytes
    - One gap length byte gives gap length = 13
    - Two verbatim bytes for tail
    - Encoding: (111)(0)(0010) 00001101 01000000 00100010

Bitmap after compression

01010100 11100010 00001101 01000000 00100010

Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical** (**integer**) **attributes**:

Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical** (**integer**) **attributes**:
  - Efficient aggregation,

Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical** (**integer**) **attributes**:
  - Efficient aggregation,
  - Efficient range filtering.

Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical (integer) attributes**:
  - Efficient aggregation,
  - Efficient range filtering.
- **Definition**:

Bit-sliced index

- **Bit-sliced index** is used for **fact table measures** and **numerical (integer) attributes**:
  - Efficient aggregation,
  - Efficient range filtering.
- **Definition**:
  - Assume, that values of attribute a are integer numbers coded by $n+1$ bits. In this case, attribute a can be stored as binary attributes $a_0, a_1, \ldots, a_n$, such that

$$a = \sum_{i=0}^{n} 2^i a_i = a_0 + 2a_1 + 2^2 a_2 \cdots + 2^n a_n.$$

    Each binary attribute a_i can be stored as bitmap index. Set of bitmap indexes of a_i, $i = 0, \ldots, n$, is the **bit-sliced index**.

- **Example**:

Amount	Bitmap
5	01**0**1
13	11**0**1
2	00**1**0
6	01**1**0
7	01**1**1

Bit-sliced index:

- B4: 01000
- B3: 11011
- B2: **00111**
- B1: 11001

Bit-sliced index

- **Example**:
  - Computing the sum:

Amount
5
13
2
6
7
Suma: 33

Bit-sliced index:	**Counting ones**:
B4: 01000	1
B3: 11011	4
B2: 00111	3
B1: 11001	3

Final results: $1 \cdot 2^3 + 4 \cdot 2^2 + 3 \cdot 2^1 + 3 \cdot 2^0 = 8 + 16 + 6 + 3 = 33$

Bit-sliced index

- **Example**:
  - Computing the sum:

Amount
5
13
2
6
7
Suma: 33

Bit-sliced index:	**Counting ones**:
B4: 01000	1
B3: 11011	4
B2: 00111	3
B1: 11001	3

Final results: $1 \cdot 2^3 + 4 \cdot 2^2 + 3 \cdot 2^1 + 3 \cdot 2^0 = 8 + 16 + 6 + 3 = 33$

Problem: How to efficiently count the number of ones in a bitmap?

Fast bitmap count

- Count the number of 1's in a bitmap:

Fast bitmap count

- Count the number of 1's in a bitmap:
  - Treat the bitmap as a byte array.
  - Pre-compute lookup table with number of 1's in each byte.
  - Cycle through bitmap one byte at a time, accumulating count using lookup table.

Fast bitmap count

- Count the number of 1's in a bitmap:
  - Treat the bitmap as a byte array.
  - Pre-compute lookup table with number of 1's in each byte.
  - Cycle through bitmap one byte at a time, accumulating count using lookup table.
- **Pseudocode**:

```
numSetBits[0] = 0;
numSetBits[1] = 1;
numSetBits[2] = 1;
numSetBits[3] = 2;
...
numSetBits[255] = 8;
count = 0;
for (int i = 0; i < n/8; i++)
     count += numSetBits[bitmap[i]];
```

Fast bitmap count

- Count the number of 1's in a bitmap:
  - Treat the bitmap as a byte array.
  - Pre-compute lookup table with number of 1's in each byte.
  - Cycle through bitmap one byte at a time, accumulating count using lookup table.
- **Pseudocode**:

```
numSetBits[0] = 0;
numSetBits[1] = 1;
numSetBits[2] = 1;
numSetBits[3] = 2;
...
numSetBits[255] = 8;
count = 0;
for (int i = 0; i < n/8; i++)
     count += numSetBits[bitmap[i]];
```

- Treating bitmap as short int array $\rightarrow$ even faster
  - Lookup table has 65536 entries instead of 256.
  - Bitmap of n bits $\rightarrow$ only add $n/16$ numbers.

Fast bitmap count

- Count the number of 1's in a bitmap
  - Use smartly properties of binary coding.
  - Making count to be linear with the number of ones.

Fast bitmap count

- Count the number of 1's in a bitmap
  - Use smartly properties of binary coding.
  - Making count to be linear with the number of ones.
- **Pseudocode**

```
word = bitmap[i];
count = 0;
while (word != 0)
     word &= (word - 1);
     count++;
```

Range filtering with bit-sliced indexes

- Bit-sliced indexes allow range filtering
- Cost of applying range predicate independent of size of range (not true for bitmap indexes or B-Trees)
- Consider an algorithm for $A < c$:
  - A is the attribute that is indexed
  - c is some constant
  - Other operations ($>$,$=$, etc.) are similar.

Range filtering with bit-sliced indexes

- **Pseudocode**:

```
set $B_{LT} = \mathbf{0}$; set $B_{EQ} = \mathbf{1}$;
for each bit slice $B_i$ from most to least signif. {
    if (bit $i$ of constant $c$ is 1) {
        $B_{LT} = B_{LT} | (B_{EQ} \& \neg B_i)$;
        $B_{EQ} = B_{EQ} \& B_i$;
    } else {
        $B_{EQ} = B_{EQ} \& \neg B_i$;
    }
}
return BLT ;
```

- **Why does it work?**
  - $B_{EQ}[j] = 1$ for all rows j that match c on the most significant bits (and only those rows);
  - A value x is less than c iff for some bit i:
    - x and c agree on all bits more significant than i,
    - and the i-th bit of x is 0, and the i-th bit of c is 1.

Range filtering with bit-sliced indexes

- **Example**:
  - Range filtering "Amount < 7" ($7 = 0111b$):

Amount	Bits
5	0101
13	1101
2	0010
6	0110
7	0111

Bit-sliced index:
B4: 01000
B3: 11011
B2: 00111
B1: 11001

B_{LT}	B_{EQ}
00000	11111

Range filtering with bit-sliced indexes

- **Example**:
  - Range filtering "Amount < 7" ($7 = 0111b$):

Amount	Bits
5	0101
13	1101
2	0010
6	0110
7	0111

Bit-sliced index:
B4: 01000
B3: 11011
B2: 00111
B1: 11001

B_{LT}	B_{EQ}
00000	11111
00000	10111

Range filtering with bit-sliced indexes

- **Example**:
  - Range filtering "Amount < 7" ($7 = 0111b$):

Amount	Bits
5	0101
13	1101
2	0010
6	0110
7	0111

Bit-sliced index:
B4: 01000
B3: 11011
B2: 00111
B1: 11001

B_{LT}	B_{EQ}
00000	11111
00000	10111
00100	10011

Range filtering with bit-sliced indexes

- **Example**:
  - Range filtering "Amount < 7" ($7 = 0111b$):

Amount	Bits
5	0101
13	1101
2	0010
6	0110
7	0111

Bit-sliced index:
B4: 01000
B3: 11011
B2: 00111
B1: 11001

B_{LT}	B_{EQ}
00000	11111
00000	10111
00100	10011
10100	00011

Range filtering with bit-sliced indexes

- **Example**:
  - Range filtering "Amount < 7" ($7 = 0111b$):

Amount	Bits
5	0101
13	1101
2	0010
6	0110
7	0111

Bit-sliced index:
B4: 01000
B3: 11011
B2: 00111
B1: 11001

B_{LT}	B_{EQ}
00000	11111
00000	10111
00100	10011
10100	00011
10110	00001

Range filtering with bit-sliced indexes

- **Example**:
  - Range filtering "Amount < 7" ($7 = 0111b$):

Amount	Bits
5	0101
13	1101
2	0010
6	0110
7	0111

Bit-sliced index:
B4: 01000
B3: 11011
B2: 00111
B1: 11001

B_{LT}	B_{EQ}
00000	11111
00000	10111
00100	10011
10100	00011
10110	00001

Projection index

- Databases usually store data in horizontal format.

Projection index

- Databases usually store data in horizontal format.
- Vertical format is more efficient for many analytical queries.

Projection index

- Databases usually store data in horizontal format.
- Vertical format is more efficient for many analytical queries.
- **Projection index** uses vertical format:

Projection index

- Databases usually store data in horizontal format.
- Vertical format is more efficient for many analytical queries.
- **Projection index** uses vertical format:
  - Logically: index entries are $< Vaule, RID >$ pairs,

Projection index

- Databases usually store data in horizontal format.
- Vertical format is more efficient for many analytical queries.
- **Projection index** uses vertical format:
  - Logically: index entries are $< Vaule, RID >$ pairs,
  - Stored in same order as records in relation (sorted by RID),

Projection index

- Databases usually store data in horizontal format.
- Vertical format is more efficient for many analytical queries.
- **Projection index** uses vertical format:
  - Logically: index entries are $< Vaule, RID >$ pairs,
  - Stored in same order as records in relation (sorted by RID),
  - In practice: storing RID is unnecessary (array storage format, array index determined from RID).

Join index

- Join indexes map the tuples in the join result of two relations to the source tables.

Product

Id	Name	Category	Join index
P1	Milk	Groceries	S1, S3, S5, S6
P2	Bread	Groceries	S2, S4

Sales

Id	Product	Customer	Date	Price
S1	P1	C1	D1	10
S2	P2	C1	D1	11
S3	P1	C2	D1	40
S4	P2	C3	D1	8
S5	P1	C2	D2	44
S6	P1	C2	D2	4

Storing and accessing multidimensional cubes

- Dense and sparse dimensions
- Organize a multi-dimensional cube by properly setting dimension types.

Storing and accessing multidimensional cubes

- Dense and sparse dimensions
- Organize a multi-dimensional cube by properly setting dimension types.
- **Example**: Assume 3 dimensions, like Product, Localization, Date and several measures like Revenue, Expenses, Netto, etc.
  - Date and measures are rather dense,
  - Product and Localization are rather sparse.
  - Two extreme data cube organizations are possible.

Storing and accessing multidimensional cubes

- **Example**: Assume 3 dimensions, like Product, Localization, Date and several measures like Revenue, Expenses, Netto, etc.
  - Two extreme data cube organizations are possible.

		JAN			FEB			MAR		
		East	West	South	East	West	South	East	West	South
	Prod. A		XXX	XXX		XXX	XXX		XXX	XXX
Rev.	Prod. B	XXX	XXX		XXX	XXX		XXX	XXX	
	Prod. C	XXX	XXX		XXX	XXX		XXX	XXX	
	Prod. A		XXX	XXX		XXX	XXX		XXX	XXX
Exp.	Prod. B	XXX	XXX		XXX	XXX		XXX	XXX	
	Prod. C	XXX	XXX		XXX	XXX		XXX	XXX	
	Prod. A		XXX	XXX		XXX	XXX		XXX	XXX
Net.	Prod. B	XXX	XXX		XXX	XXX		XXX	XXX	
	Prod. C	XXX	XXX		XXX	XXX		XXX	XXX	

Storing and accessing multidimensional cubes

- **Example**: Assume 3 dimensions, like Product, Localization, Date and several measures like Revenue, Expenses, Netto, etc.
  - Two extreme data cube organizations are possible.

		East			West			South		
		JAN	FEB	MAR	JAN	FEB	MAR	JAN	FEB	MAR
	Rev.				XXX	XXX	XXX	XXX	XXX	XXX
Prod. A	Exp.				XXX	XXX	XXX	XXX	XXX	XXX
	Net.				XXX	XXX	XXX	XXX	XXX	XXX
	Rev.	XXX	XXX	XXX	XXX	XXX	XXX			
Prod. B.	Exp.	XXX	XXX	XXX	XXX	XXX	XXX			
	Net.	XXX	XXX	XXX	XXX	XXX	XXX			
	Rev.	XXX	XXX	XXX	XXX	XXX	XXX			
Prod. C.	Exp.	XXX	XXX	XXX	XXX	XXX	XXX			
	Net.	XXX	XXX	XXX	XXX	XXX	XXX			

- **Example**: Assume 3 dimensions, like Product, Localization, Date and several measures like Revenue, Expenses, Netto, etc.
  - Two extreme data cube organizations are possible.
    - The second organization allows to efficiently store the cube using 3×3 data chunks — some of the chunks are empty.
    - The first organization is inefficient.

Storing and accessing multidimensional cubes

- Construct an index on sparse dimensions.

Storing and accessing multidimensional cubes

- Construct an index on sparse dimensions.
- Each leaf points to a multidimensional array that stores dense dimensions.

Storing and accessing multidimensional cubes

- Construct an index on sparse dimensions.
- Each leaf points to a multidimensional array that stores dense dimensions.
- The multidimensional arrays can be still compressed: bitmap compression, run-length encoding, etc.

Compression

- **Example**:
  - A sparse array:

	Product	Mountain	Road	Touring
Day	1/1/2010			3
	2/1/2011		2	
	3/1/2011			5

can be stored as a sequence of non-missing values

$$3, 2, 5$$

- **Example**:
  - A sparse array:

	Product	Mountain	Road	Touring
Day	1/1/2010			3
	2/1/2011		2	
	3/1/2011			5

can be stored as a sequence of non-missing values

$$3, 2, 5,$$

but we need add additional information about positions of these values:

Compression

- **Example**:
  - A sparse array:

	Product	Mountain	Road	Touring
Day	1/1/2010			3
	2/1/2011		2	
	3/1/2011			5

can be stored as a sequence of non-missing values

$$3, 2, 5,$$

but we need add additional information about positions of these values:

- Indexes: 3,5,9
- Gaps: 2,1,3
- Bitmaps: 001010001
- Run-length codes: Null, Null, 3, Null, 2, Null×3, 5
- Indexes and gaps can be further coded by prefix codes.

Query processing

- Rewriting a query into an equivalent form so that it is less expensive to evaluate and finding a plan for evaluating the query that incurs minimal cost are classical database problems.

Query processing

- Rewriting a query into an equivalent form so that it is less expensive to evaluate and finding a plan for evaluating the query that incurs minimal cost are classical database problems.
- Data warehouses having a well-defined structure allow one to apply a broad spectrum of optimization techniques.

Query processing

- Typical query to data warehouse:

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,
  - Filter condition on dimensions,

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,
  - Filter condition on dimensions,
  - Grouping and aggregation.

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,
  - Filter condition on dimensions,
  - Grouping and aggregation.
- Traditional processing for such queries:

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,
  - Filter condition on dimensions,
  - Grouping and aggregation.
- Traditional processing for such queries:
  - Join,

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,
  - Filter condition on dimensions,
  - Grouping and aggregation.
- Traditional processing for such queries:
  - Join,
  - Filtering,

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,
  - Filter condition on dimensions,
  - Grouping and aggregation.
- Traditional processing for such queries:
  - Join,
  - Filtering,
  - Grouping,

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,
  - Filter condition on dimensions,
  - Grouping and aggregation.
- Traditional processing for such queries:
  - Join,
  - Filtering,
  - Grouping,
  - Aggregation.

Query processing

- Typical query to data warehouse:
  - Joins of the fact table with the dimensions,
  - Filter condition on dimensions,
  - Grouping and aggregation.
- Traditional processing for such queries:
  - Join,
  - Filtering,
  - Grouping,
  - Aggregation.
- Choice of the join algorithm and query processing strategy has large impact on query cost.

Dimension Cartesian product

- Consider a scenario in which:

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,
  - 3 dimension tables, each with 100 rows

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,
  - 3 dimension tables, each with 100 rows
  - Filters select 10 rows from each dimension

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,
  - 3 dimension tables, each with 100 rows
  - Filters select 10 rows from each dimension
- Traditional processing for such a query:

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,
  - 3 dimension tables, each with 100 rows
  - Filters select 10 rows from each dimension
- Traditional processing for such a query:
  - Join fact to dimension A: Produce intermediate result with 10 million rows

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,
  - 3 dimension tables, each with 100 rows
  - Filters select 10 rows from each dimension
- Traditional processing for such a query:
  - Join fact to dimension A: Produce intermediate result with 10 million rows
  - Join result to dimension B: Produce intermediate result with 1 million rows

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,
  - 3 dimension tables, each with 100 rows
  - Filters select 10 rows from each dimension
- Traditional processing for such a query:
  - Join fact to dimension A: Produce intermediate result with 10 million rows
  - Join result to dimension B: Produce intermediate result with 1 million rows
  - Join result to dimension C: Produce intermediate result with 100 000 rows

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,
  - 3 dimension tables, each with 100 rows
  - Filters select 10 rows from each dimension
- Traditional processing for such a query:
  - Join fact to dimension A: Produce intermediate result with 10 million rows
  - Join result to dimension B: Produce intermediate result with 1 million rows
  - Join result to dimension C: Produce intermediate result with 100 000 rows
  - Perform grouping and aggregation

Dimension Cartesian product

- Consider a scenario in which:
  - Fact table has 100 million rows,
  - 3 dimension tables, each with 100 rows
  - Filters select 10 rows from each dimension
- Traditional processing for such a query:
  - Join fact to dimension A: Produce intermediate result with 10 million rows
  - Join result to dimension B: Produce intermediate result with 1 million rows
  - Join result to dimension C: Produce intermediate result with 100 000 rows
  - Perform grouping and aggregation
- **Drawbacks**: Each join is expensive and intermediate results are quite large!

Dimension Cartesian product

- Alternatively, one can perform Cartesian product over dimensions:

Dimension Cartesian product

- Alternatively, one can perform Cartesian product over dimensions:
  - Join dimensions A and B: Result is Cartesian product of all combinations, i.e., 100 rows (10 A rows $\times$ 10 B rows)

Dimension Cartesian product

- Alternatively, one can perform Cartesian product over dimensions:
  - Join dimensions A and B: Result is Cartesian product of all combinations, i.e., 100 rows (10 A rows $\times$ 10 B rows)
  - Join Cartesian product of A and B to dimension C: another Cartesian product with 1000 rows (10 A rows $\times$ 10 B rows $\times$ 10 C rows)

Dimension Cartesian product

- Alternatively, one can perform Cartesian product over dimensions:
  - Join dimensions A and B: Result is Cartesian product of all combinations, i.e., 100 rows (10 A rows $\times$ 10 B rows)
  - Join Cartesian product of A and B to dimension C: another Cartesian product with 1000 rows (10 A rows $\times$ 10 B rows $\times$ 10 C rows)
  - Join Cartesian product of A, B, and C to fact table: Produce intermediate result with 100,000 rows

Dimension Cartesian product

- Alternatively, one can perform Cartesian product over dimensions:
  - Join dimensions A and B: Result is Cartesian product of all combinations, i.e., 100 rows (10 A rows $\times$ 10 B rows)
  - Join Cartesian product of A and B to dimension C: another Cartesian product with 1000 rows (10 A rows $\times$ 10 B rows $\times$ 10 C rows)
  - Join Cartesian product of A, B, and C to fact table: Produce intermediate result with 100,000 rows
  - Perform grouping and aggregation

Dimension Cartesian product

- Pros:

Dimension Cartesian product

- Pros:
  - Computing Cartesian product is cheap: Few rows in dimension tables

Dimension Cartesian product

- Pros:
  - Computing Cartesian product is cheap: Few rows in dimension tables
  - Only one expensive join rather than three

Dimension Cartesian product

- Pros:
  - Computing Cartesian product is cheap: Few rows in dimension tables
  - Only one expensive join rather than three
- Cons:

Dimension Cartesian product

- Pros:
  - Computing Cartesian product is cheap: Few rows in dimension tables
  - Only one expensive join rather than three
- Cons:
  - Only applicable for a small number of dimensions,

Dimension Cartesian product

- Pros:
  - Computing Cartesian product is cheap: Few rows in dimension tables
  - Only one expensive join rather than three
- Cons:
  - Only applicable for a small number of dimensions,
  - and a small number of rows in each dimension satisfying filters

Early aggregation

- Sometimes *group by* can be handled in two phases:

Early aggregation

- Sometimes *group by* can be handled in two phases:
  - Perform partial aggregation early as a data reduction technique

Early aggregation

- Sometimes *group by* can be handled in two phases:
  - Perform partial aggregation early as a data reduction technique
  - Finish up the aggregation after completing all joins

Early aggregation

- Sometimes *group by* can be handled in two phases:
  - Perform partial aggregation early as a data reduction technique
  - Finish up the aggregation after completing all joins
- Advantages:

Early aggregation

- Sometimes *group by* can be handled in two phases:
  - Perform partial aggregation early as a data reduction technique
  - Finish up the aggregation after completing all joins
- Advantages:
  - Early grouping and aggregation can reduce the size of intermediate results.

Early aggregation

- Sometimes *group by* can be handled in two phases:
  - Perform partial aggregation early as a data reduction technique
  - Finish up the aggregation after completing all joins
- Advantages:
  - Early grouping and aggregation can reduce the size of intermediate results.
  - The scan for performing the join can be exploited for the grouping.

Early aggregation

- **Example**:

```
SELECT st.district, sum(s.price)
FROM Sales s, Store st, Data d
WHERE s.store id = st.id AND s.date id = d.id AND d.year
= 2003
GROUP BY st.district;
```

  - Assumptions:
    - Sales fact has 100 million rows
    - Store dimension has 100 rows
    - Date dimension has 1000 rows (365 in 2003)

Early aggregation

- Traditional evaluation for this query:

Early aggregation

- Traditional evaluation for this query:
  - Join of the fact table `Sales` with the dimension tables `Date`, filtering based on `Year`: Result has 36.5 million rows,

Early aggregation

- Traditional evaluation for this query:
  - Join of the fact table `Sales` with the dimension tables `Date`, filtering based on `Year`: Result has 36.5 million rows,
  - Join result with `Store`: Result has 36.5 million rows

Early aggregation

- Traditional evaluation for this query:
  - Join of the fact table `Sales` with the dimension tables `Date`, filtering based on `Year`: Result has 36.5 million rows,
  - Join result with `Store`: Result has 36.5 million rows
  - Group by `st.district` and compute aggregates.

Early aggregation

- Traditional evaluation for this query:
  - Join of the fact table `Sales` with the dimension tables `Date`, filtering based on `Year`: Result has 36.5 million rows,
  - Join result with `Store`: Result has 36.5 million rows
  - Group by `st.district` and compute aggregates.
- Better strategy relies on:

Early aggregation

- Traditional evaluation for this query:
  - Join of the fact table `Sales` with the dimension tables `Date`, filtering based on `Year`: Result has 36.5 million rows,
  - Join result with `Store`: Result has 36.5 million rows
  - Group by `st.district` and compute aggregates.
- Better strategy relies on:
  - Group sales by `(store id, date id)` and compute aggregates: Result has 100 000 rows

Early aggregation

- Traditional evaluation for this query:
  - Join of the fact table `Sales` with the dimension tables `Date`, filtering based on `Year`: Result has 36.5 million rows,
  - Join result with `Store`: Result has 36.5 million rows
  - Group by `st.district` and compute aggregates.
- Better strategy relies on:
  - Group sales by `(store id, date id)` and compute aggregates: Result has 100 000 rows
  - Join result with `Date` dimension, filtered based on `Year`: Result has 36 500 rows

Early aggregation

- Traditional evaluation for this query:
  - Join of the fact table `Sales` with the dimension tables `Date`, filtering based on `Year`: Result has 36.5 million rows,
  - Join result with `Store`: Result has 36.5 million rows
  - Group by `st.district` and compute aggregates.
- Better strategy relies on:
  - Group sales by `(store id, date id)` and compute aggregates: Result has 100 000 rows
  - Join result with `Date` dimension, filtered based on `Year`: Result has 36 500 rows
  - Join result with `Store` dimension: Result has 36 500 rows

Early aggregation

- Traditional evaluation for this query:
  - Join of the fact table `Sales` with the dimension tables `Date`, filtering based on `Year`: Result has 36.5 million rows,
  - Join result with `Store`: Result has 36.5 million rows
  - Group by `st.district` and compute aggregates.
- Better strategy relies on:
  - Group sales by `(store id, date id)` and compute aggregates: Result has 100 000 rows
  - Join result with `Date` dimension, filtered based on `Year`: Result has 36 500 rows
  - Join result with `Store` dimension: Result has 36 500 rows
  - Group by `District` and compute aggregates.

Early aggregation

- Pros:

Early aggregation

- Pros:
  - Initial aggregation can be fast with appropriate index

Early aggregation

- Pros:
  - Initial aggregation can be fast with appropriate index
  - Result of early aggregation significantly smaller than fact table: fewer rows and fewer columns

Early aggregation

- Pros:
  - Initial aggregation can be fast with appropriate index
  - Result of early aggregation significantly smaller than fact table: fewer rows and fewer columns
  - Joins to dimension tables are cheaper (intermediate result is much smaller than fact table)

Early aggregation

- Pros:
  - Initial aggregation can be fast with appropriate index
  - Result of early aggregation significantly smaller than fact table: fewer rows and fewer columns
  - Joins to dimension tables are cheaper (intermediate result is much smaller than fact table)
- Cons:

Early aggregation

- Pros:
  - Initial aggregation can be fast with appropriate index
  - Result of early aggregation significantly smaller than fact table: fewer rows and fewer columns
  - Joins to dimension tables are cheaper (intermediate result is much smaller than fact table)
- Cons:
  - Cannot take advantage of data reduction due to filters

Early aggregation

- Pros:
  - Initial aggregation can be fast with appropriate index
  - Result of early aggregation significantly smaller than fact table: fewer rows and fewer columns
  - Joins to dimension tables are cheaper (intermediate result is much smaller than fact table)
- Cons:
  - Cannot take advantage of data reduction due to filters
  - Two aggregation steps instead of one

Outline

Motivation

- Computational burden $\rightarrow$ divide and conquer

Motivation

- Computational burden $\rightarrow$ divide and conquer
  - Data partitioning

Motivation

- Computational burden $\rightarrow$ divide and conquer
  - Data partitioning
  - Distributed systems

Data partitioning

- In general, partitioning divides tables and indexes into a smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity.

Data partitioning

- In general, partitioning divides tables and indexes into a smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity.
- Partitioning concerns tables in distributed systems like MapReduce (sometimes referred to as sharding), distributed and parallel databases, but also conventional tables.

Data partitioning

- In general, partitioning divides tables and indexes into a smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity.
- Partitioning concerns tables in distributed systems like MapReduce (sometimes referred to as sharding), distributed and parallel databases, but also conventional tables.
- Partitioning can provide benefits by improving manageability, performance, and availability.

Data partitioning

- In general, partitioning divides tables and indexes into a smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity.
- Partitioning concerns tables in distributed systems like MapReduce (sometimes referred to as sharding), distributed and parallel databases, but also conventional tables.
- Partitioning can provide benefits by improving manageability, performance, and availability.
- Partitioning is transparent for database queries.

Data partitioning

- In general, partitioning divides tables and indexes into a smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity.
- Partitioning concerns tables in distributed systems like MapReduce (sometimes referred to as sharding), distributed and parallel databases, but also conventional tables.
- Partitioning can provide benefits by improving manageability, performance, and availability.
- Partitioning is transparent for database queries.
- Horizontal vs. vertical vs. chunk partitioning.

Data partitioning

- Table or index is subdivided into smaller pieces.

Data partitioning

- Table or index is subdivided into smaller pieces.
- Each piece of database object is called a partition.

Data partitioning

- Table or index is subdivided into smaller pieces.
- Each piece of database object is called a partition.
- Each partition has its own name, and may have its own storage characteristics (e.g. table compression).

Data partitioning

- Table or index is subdivided into smaller pieces.
- Each piece of database object is called a partition.
- Each partition has its own name, and may have its own storage characteristics (e.g. table compression).
- From the perspective of a database administrator, a partitioned object has multiple pieces which can be managed either collectively or individually.

Data partitioning

- Table or index is subdivided into smaller pieces.
- Each piece of database object is called a partition.
- Each partition has its own name, and may have its own storage characteristics (e.g. table compression).
- From the perspective of a database administrator, a partitioned object has multiple pieces which can be managed either collectively or individually.
- From the perspective of the application, however, a partitioned table is identical to a non-partitioned table.

Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.

Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:

Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:
  - Hash partitioning: Rows divided into partitions using a hash function

Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:
  - Hash partitioning: Rows divided into partitions using a hash function
  - Range partitioning: Each partition holds a range of attribute values

Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:
  - Hash partitioning: Rows divided into partitions using a hash function
  - Range partitioning: Each partition holds a range of attribute values
  - List partitioning: Rows divided according to lists of values that describe the partition

Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:
  - Hash partitioning: Rows divided into partitions using a hash function
  - Range partitioning: Each partition holds a range of attribute values
  - List partitioning: Rows divided according to lists of values that describe the partition
  - Composite Partitioning: partitions data using the range method, and within each partition, subpartitions it using the hash or list method.

Data partitioning

- **Example**:

```
CREATE TABLE sales_list (
  salesman_id NUMBER(5),
  salesman_name VARCHAR2(30),
  sales_state VARCHAR2(20),
  sales_amount NUMBER(10),
  sales_date DATE)
  PARTITION BY LIST(sales_state)
  (
    PARTITION sales_west VALUES('California', 'Hawaii'),
    PARTITION sales_east VALUES ('New York', 'Virginia'),
    PARTITION sales_central VALUES('Texas', 'Illinois')
    PARTITION sales_other VALUES(DEFAULT)
  )
);
```

Partitioning and sorting

- External-memory sorting:

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k+1$ units (k input and one output buffer).

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k + 1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k + 1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k + 1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k+1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k + 1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k+1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first k/n of data from each sorted partition to main memory (use all k input buffers).

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k+1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first k/n of data from each sorted partition to main memory (use all k input buffers).
  - Do

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k+1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first k/n of data from each sorted partition to main memory (use all k input buffers).
  - Do
    - Perform k-way merge sort using the output buffer to store globally sorted data.

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k + 1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first k/n of data from each sorted partition to main memory (use all k input buffers).
  - Do
    - Perform k-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k+1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first k/n of data from each sorted partition to main memory (use all k input buffers).
  - Do
    - Perform k-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.
    - If the ith input buffer is exhausted, read next portion from ith partition.

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k+1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first k/n of data from each sorted partition to main memory (use all k input buffers).
  - Do
    - Perform k-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.
    - If the ith input buffer is exhausted, read next portion from ith partition.
- Remark that $k \geq \sqrt{n}$ (otherwise we need additional merge passes).

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k + 1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first k/n of data from each sorted partition to main memory (use all k input buffers).
  - Do
    - Perform k-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.
    - If the ith input buffer is exhausted, read next portion from ith partition.
- Remark that $k \geq \sqrt{n}$ (otherwise we need additional merge passes).
- External-memory sorting is used in merge-join of large data sets.

Partitioning and sorting

- External-memory sorting:
  - Let data be of size n and main memory be of size $k+1$ units (k input and one output buffer).
  - Partition data into n/k parts (does not have to be made explicitly).
  - For each partition (each uses k memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first k/n of data from each sorted partition to main memory (use all k input buffers).
  - Do
    - Perform k-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.
    - If the ith input buffer is exhausted, read next portion from ith partition.
- Remark that $k \geq \sqrt{n}$ (otherwise we need additional merge passes).
- External-memory sorting is used in merge-join of large data sets.
- Similarly one can generalize hash-join to the so-called partitioned hash-join.

Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,

Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,
- Partial compression,

Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,
- Partial compression,
- Partial backups,

Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,
- Partial compression,
- Partial backups,
- Data recovery can concern partitions,

Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,
- Partial compression,
- Partial backups,
- Data recovery can concern partitions,
- "Divide and conquer" approach to data management.

- Partition fact table:

Data partitioning and data warehouses

- Partition fact table:
  - Fact tables are big,

Data partitioning and data warehouses

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,

Data partitioning and data warehouses

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,

Data partitioning and data warehouses

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.

Data partitioning and data warehouses

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:

Data partitioning and data warehouses

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,

Data partitioning and data warehouses

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,
  - Storing multiple copies of them is cheap,

Data partitioning and data warehouses

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,
  - Storing multiple copies of them is cheap,
  - No communication needed for parallel joins

Data partitioning and data warehouses

- One big dimension:

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)
  - Partition the big dimension table

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)
  - Partition the big dimension table
  - Partition fact table on key of big dimension

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)
  - Partition the big dimension table
  - Partition fact table on key of big dimension
- Reducing load time via partitioning

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)
  - Partition the big dimension table
  - Partition fact table on key of big dimension
- Reducing load time via partitioning
  - Often fact tables are partitioned on Date

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)
  - Partition the big dimension table
  - Partition fact table on key of big dimension
- Reducing load time via partitioning
  - Often fact tables are partitioned on Date
  - Also indexes, aggregate tables, etc.

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)
  - Partition the big dimension table
  - Partition fact table on key of big dimension
- Reducing load time via partitioning
  - Often fact tables are partitioned on Date
  - Also indexes, aggregate tables, etc.
  - Newly loaded records go into the last partition

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)
  - Partition the big dimension table
  - Partition fact table on key of big dimension
- Reducing load time via partitioning
  - Often fact tables are partitioned on Date
  - Also indexes, aggregate tables, etc.
  - Newly loaded records go into the last partition
  - Only indexes and aggregates for that partition need to be updated

Data partitioning and data warehouses

- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer)
  - Partition the big dimension table
  - Partition fact table on key of big dimension
- Reducing load time via partitioning
  - Often fact tables are partitioned on Date
  - Also indexes, aggregate tables, etc.
  - Newly loaded records go into the last partition
  - Only indexes and aggregates for that partition need to be updated
  - All other partitions remain unchanged

Data partitioning and data warehouses

- Expiring old data

Data partitioning and data warehouses

- Expiring old data
  - Often older data is less useful / relevant for data analysts

Data partitioning and data warehouses

- Expiring old data
  - Often older data is less useful / relevant for data analysts
  - To reduce data warehouse size, old data is often deleted

Data partitioning and data warehouses

- Expiring old data
  - Often older data is less useful / relevant for data analysts
  - To reduce data warehouse size, old data is often deleted
  - If data is partitioned on date, simply delete or compress the oldest partition

Data partitioning and data warehouses

- Expiring old data
  - Often older data is less useful / relevant for data analysts
  - To reduce data warehouse size, old data is often deleted
  - If data is partitioned on date, simply delete or compress the oldest partition
- Multi-table joins

Data partitioning and data warehouses

- Expiring old data
  - Often older data is less useful / relevant for data analysts
  - To reduce data warehouse size, old data is often deleted
  - If data is partitioned on date, simply delete or compress the oldest partition
- Multi-table joins
  - Join can be applied with two tables partitioned on the join key,

Data partitioning and data warehouses

- Expiring old data
  - Often older data is less useful / relevant for data analysts
  - To reduce data warehouse size, old data is often deleted
  - If data is partitioned on date, simply delete or compress the oldest partition
- Multi-table joins
  - Join can be applied with two tables partitioned on the join key,
  - A large join is then broken into smaller joins that occur between each of the partitions, completing the overall join in less time.

Outline

Summary

- Physical storage,
- Denormalization and summarization,
- Data access,
- Data partitioning.

Bibliography

- J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, second edition edition, 2006
- `https://graphics.stanford.edu/~seander/bithacks.html`