CO572 - Advanced Databases (60002)

Lecture 1

What is a **database management system**? A **database** is any structured collection of data points, which can be a relational table, a set, a vector, a graph, or anything along those lines. **Data management** is needed for **data-intensive applications**, we say something processes a significant amount of data if the amount of data is larger than what fits in the CPU's cache, something in the order of a few MB. We can say that below this threshold (around 5MB - 50MB), there are other factors that likely dominate performance. A **system** is made up from components that interact together to achieve a greater goal and is usually applicable to many situations.

Applications

- The scenario is at a hospital. At any given time, there are 800 patients, producing a sample per second of 5 metrics. There are also 200 doctors and nurses, who each produce a textual report every 10 minutes, and 80 lab technicians producing a structured dataset of 10 metrics every 5 minutes. Everything must be stored **reliably**, it cannot be lost after it is stored (or with a probability $p < 0.001$).
- You are developing a interactive dashboard for a global retail company. This company has stored 500GBs of sales, inventory, and customer records, and shall provide interactive access to calculated statistics. This should allow filtering of the dataset with predicates, and support the calculation of the sums of records, all with response times below a second.

Below are some examples of typical data-intensive application patterns;

- Online Transaction Processing **OLTP**
  - lots of small updates to a persistent database
  - focused on throughput (do as many updates as possible in a time-frame)
  - ACID is key (reliable)
- Online Analytical Processing **OLAP**
  - running a single data analysis task
  - focus on latency (do queries as quickly as possible)
  - **ad-hoc** queries - we don't know what they'll be
- Reporting
  - running many analysis tasks in a fixed time budget
  - focused on resource efficiency - if we can do the same task by the same time it's due with fewer resources, then it would be cheaper
  - queries are known in advance (and can be compiled into the system)
- Hybrid Transactional / Analytical Processing **HTAP**
  - a mix of **OLTP** and **OLAP**
  - small updates woven with larger analytics
  - common application is fraud detection

Data-intensive Application vs Management System

A **application** is not generic - it's often domain-specific with the logic baked in, and is therefore hard to generalise to other applications. Generally, the cost (such as adapting for other applications) of application-specific data management outweighs the benefits. We can generalise the following (from an application to a management system);

- *Yelp*
- mobile app for geo-services
- library to manage unordered collections of tagged coordinates
- spatial data management library
- relational database
- block storage system

Requirements of a Data Management System

A data management system should fulfil the following requirements;

- **efficiency** should not be significantly slower than hand written applications
- **resilience** should recover from problems, such as power outage, hardware or software failures
- **robustness**

  should have predictable performance; a small change in the query should not lead to major changes in performance
- **scalability**

  should make efficient use of available resources - increase in resources should lead to an improvement of performance
- **concurrency**

  should transparently serve multiple clients transparently (without impacting results)

Solutions

DBMSs often provide some ingenious solutions;

- **Physical and Logical Data Model Separation**

  We typical provide a **logical data model** to the user, as they often send data in a **fire and forget** manner. The user doesn't typically care about file format, storage devices, nor portability. The DMBSs can then separate external from the internal model and therefore exploit these degrees of freedom for performance. For example, if the user doesn't care where the data is stored, we can keep the hot data on an SSD and the cold data on disk or even tape.
- **Transparent Concurrency: Transactional Semantics**
  - **Atomic** run completely or not at all (if aborted, everything is reverted)
  - **Consistent** constraints must hold before and after (can be inconsistent between)
  - **Isolated** run like you were alone on the system
  - **Durable** after a transaction is committed **nothing** can undo it
- **Ease of Use: Declarative Data Analysis**

  Retrieving information about data should be done by describing the result and the system will generate it. This can be a single tuple, some statistics, a detailed generated report, or even training a model to predict data.

However, they are not to be used as a filesystem. Similarly, it cannot be used as runtime for applications - there are support for user-defined functions, but should not be used a such. They also should not be used to store intermediate data (such as whether a user is logged in).

Relational Algebra

A schema is the definition of the attributes of the tuples in the relations, but also can contain integrity constraints.

• **vector**	ordered collection of objects of the same type
• **tuple**	ordered collection of objects of different type
• **bag**	unordered collection of objects of the same type
• **set**	unordered collection of unique objects of the same type

Relational algebra is used to define the semantics of operations and is used for logical optimisation. However, it's not actually that useful for end-users. A relation is an array which represents an n-ary relation R, with the following properties;

- each row represents an n-tuple of R
- the order of rows doesn't matter
- all rows are distinct (combined with above is a set)
- the order of columns matters - all rows have the same schema
- each column has a label (defines the schema of our relation)

Relations are **almost** sets of tuples. A rough implementation in C++ is as follows;

```cpp
template <typename... types>
struct Relation {
    using OutputType = tuple<types...>;
    set<tuple<types...>> data;
    array<string, sizeof...(types)> schema;
    Relation(){};
    Relation(array<string, sizeof...(types)> schema, set<tuple<types...>> data):
        schema(schema), data(data) {}
};
```

A relation can then be written as follows;

```cpp
auto createCustomerTable() {
    Relation<int, string, string> customer(
        {"ID", "Name", "ShippingAddress"}, // labels of the attribute
        {{1, "james", "address 1"},
         {2, "steve", "another address}});
    return customer;
}
```

A **relational expression** is composed from **relational operators**, and will often be referred to as a **(logical) plan**. **Cardinality** is the number of tuples in a set. Relational operations are set-based, and therefore order-invariant and duplicates are eliminated. Additionally, it's **closed**;

- every operator produces a relation as an output
- every operator accepts one or two relations as input
- simplifies the composition of operators into expressions (however expressions can be invalid)

Relational operators can be implemented as follows;

```
template <typename... types> struct Operator : public Relation<types...> {}; //
    therefore an operator is a relation
```

A minimal set of relational operators is as follows;

- **project** (π)
  - extract one or more attributes from a relation
  - preserves relational semantics
  - changes schema

  For example, given a table;

  table1

field1	field2	field3
A	B	C
D	E	F
G	E	I

  $\pi_{\texttt{field2}}$table1

field2
B
E

  The cardinality of the output of a projection can only be determined by evaluating it, as duplicates can be eliminated. On the other hand, the upper bound of the cardinality of the output is the cardinality of the input.

  It can also extract one or more attributes from a relation and perform a scalar operation on them.

```
template <typename InputOperator, typename... outputTypes>
struct Project : public Operator<outputTypes...> {
    InputOperator input;

    variant<function<tuple<outputTypes...>(typename InputOperator::OutputType
       )>,
            set<pair<string, string>>>
        projections;

    Project(InputOperator input, function<tuple<outputTypes...>(typename
       InputOperator::OutputType)> projections : input(input), projections(
       projections) {};

    Project(InputOperator input, set<pair<string, string>> projections :
       input(input), projections(projections) {};
}

void projectionExample {
    auto customer = createCustomerTable();

    auto p1 = Project<decltype(customer), string>(customer, [](auto input) {
       return get<1>(input); });

    auto p2 = Project<decltype(customer), string>(customer, {{"Name", "
       customerName"}});
}
```

- **select** (σ)
  - produces a new relation containing tuples which satisfy a condition
  - does not change schema

  - changes cardinality (number of tuples in a relation)

For example, given a table;

table1		
field1	field2	field3
A	B	C
D	E	F
G	E	I

$\sigma_{\texttt{field2=E}}$table1		
field1	field2	field3
D	E	F
G	E	I

The cardinality can only be determined by evaluating it, and the upper bound of the cardinality is also the cardinality of the input.

```
enum class Comparator { less, lessEqual, equal, greaterEqual, greater };

struct Column {
    string name;
    Column(string name) : name(name) {};
}
using Value = variant<string, int float>;

struct Condition {
    Column leftHandSide;
    Comparator compare;
    variant<Column, Value> rightHandSide;

    Condition(Column leftHandSide, Comparator compare, variant<Column, Value>
        rightHandSide): leftHandSide(leftHandSide), compare(compare),
       rightHandSide(rightHandSide) {};
}
```

- **cross product** ($\times$)
  - takes two inputs
  - produces a new relation by combining every tuple from the left with every tuple from the right
  - changes the schema

For example, given tables;

table1	
field1	field2
A	B
G	E

table2	
fieldA	fieldB
2	6
5	1

table1 $\times$ table2			
field1	field2	fieldA	fieldB
A	B	2	6
A	B	5	1
G	E	2	6
G	E	5	1

The cardinality of the output is the product of the two input cardinalities.

```
template <typename LeftInputOperator, typename RightInputOperator>
struct CrossProduct : public Operator<Concat<typename LeftInputOperator::
    OutputType, typename RightOutputOperator::OutputType>> {
    LeftInputOperator leftInput;
    RightInputOperator rightInput;
    CrossProduct(LeftInputOperator leftInput, RightInputOperator  rightInput)
        : leftInput(leftInput), rightInput(rightInput) {};
};
```

- **union** ($\cup$)
  - produces a new relation from two relations containing any tuple that is present in either
  - does not change the schema (but does require schema compatibility)
  - changes cardinality

  No example provided, because it's quite simple.

```
template <typename LeftInputOperator, typename RightInputOperator>
struct Union : public Operator<typename LeftInputOperator:::OutputType> {
    LeftInputOperator leftInput;
    RightInputOperator rightInput;

    Union(LeftInputOperator leftInput, RightInputOperator rightInput):
       leftInput(leftInput), rightInput(rightInput){};
};
```

  The cardinality can only be known by evaluating (due to duplicates), and the upper bound is the sum of the cardinalities of the input.

- **difference** ($-$)
  - produces new relation from two relations containing tuples present in the first but not the second
  - doesn't change schema (requires compatibility)
  - changes cardinality

  No example provided, because it's quite simple.

```
template <typename LeftInputOperator, typename RightInputOperator>
struct Difference : public Operator<typename LeftInputOperator:::OutputType>
   {
    LeftInputOperator leftInput;
    RightInputOperator rightInput;

    Difference(LeftInputOperator leftInput, RightInputOperator rightInput):
       leftInput(leftInput), rightInput(rightInput){};
};
```

- **group aggregation** (Γ)
  - produces new relation from one input by grouping tuples that have equal values in some attributes and aggregate others - groups are defined by the set of grouping attributes (can be empty), and the aggregates are defined by the set of **aggregations** which are triples consisting of;
    - **input** attribute
    - **aggregation function** (min, max, avg, sum, count)
    - **output** attribute
  - this changes both the schema and cardinality

  For example, given the following table;

customer

ID	Name	City
1	james	London
2	steve	London
3	kate	Manchester

$\Gamma((\texttt{City}), ((\texttt{ID}, \texttt{count}, \texttt{c})))\texttt{Customer}$

City	c
London	2
Manchester	1

```cpp
enum class AggregationFunction { min, max, sum, avg, count };

template <typename InputOperator, typename... Output>
struct GroupedAggregation : public Operator<Output...> {
    InputOperator input;
    set<string> groupAttributes;
    set<tuple<string, AggregationFunction, string>> aggregations;
    GroupedAggregation(InputOperator input, set<string> groupAttributes, set<
       tuple<string, AggregationFunction, string>> aggregations): input(input
       ), groupAttributes(groupAttributes), aggregations(aggregations){};
};
```

- **Top-N** (T)
  - produce new relation from one input selecting the tuples with the N greatest values with respect to an attribute
  - changes cardinality, but maintains schema

For example, given the following table;

customer

ID	Name	City
1	james	London
2	steve	London
3	kate	Manchester

$T_{(2,\texttt{ID})}$Customer

ID	Name	City
2	steve	London
3	kate	Manchester

```cpp
template <typename InputOperator>
struct TopN : public Operator<typename InputOperator::OutputType> {
    InputOperator input;
    size_t N;
    string predicate;
    TopN(InputOperator input, size_t N, string predicate): input(input), N(N)
       , predicate(predicate){};
};
```

Since relational algebra is closed, operators can be combined as long as signatures are respected (cross product takes two inputs, whereas selections and projections take one);

$$\pi_{\texttt{BookID}}(\sigma_{\texttt{Order.ID == OrderedItem.OrderID}}(\texttt{Order} \times \texttt{OrderedItem}))$$