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;

```
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;

```
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

```
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

```
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){};
};
```

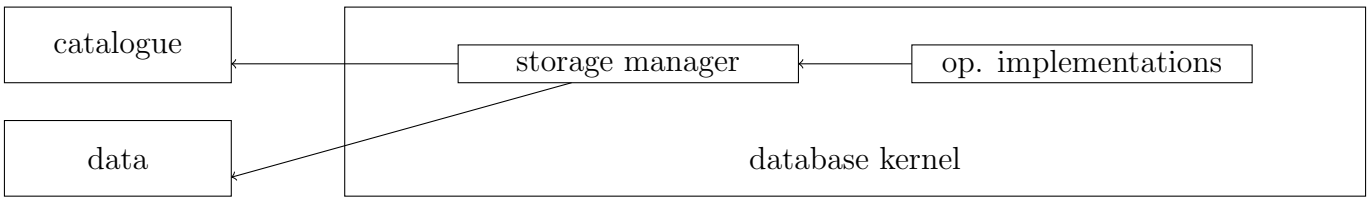
Since relational algebra is closed, operators can be combined as long as signatures are respected (cross product takes two inputs, whereas selections and projections take one);

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

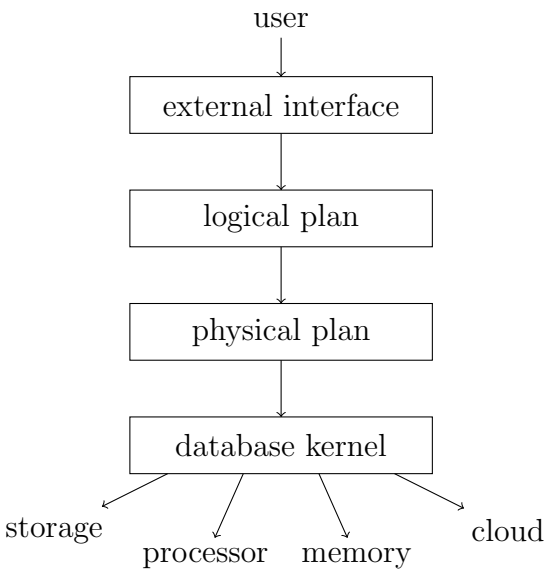
Lecture 2

Architecture

The logical plan is the relational algebra discussed last lecture and the database kernel actually interacts with the resources mentioned. A database kernel is a library of core functionality, including I/O, memory management, operators, etc. Generally, it provides an interface to subsystems (similar to an OS kernel). A basic database kernel architecture is as follows (assuming both catalogue and data live in memory).



The DBMS architecture is as follows;



The kernel interface, in C++, would look like the following;

```
class StorageManager;
class OperatorImplementations;

// just provides an interface to the above
class DatabaseKernel {
    StorageManager& getStorageManager();
    OperatorImplementations& getOperatorImplementations();
}
```

Storage

The first operation every database needs to support inserting new tuples. These inserts are usually not optimised (generally) - they arrive at the storage layer as intact tuples. Consider the following example, in SQL;

```
CREATE TABLE Employee (
    id int,
    name varchar,
    salary int,
    joiningDate int
);

INSERT INTO Employee VALUES(1, 'james', 100000, 43429342);
SELECT * FROM Employee WHERE id=1;
DELETE FROM Employee WHERE name='james';
```

The equivalent in C++ would be as follows;

```
StorageManager().getTable("Employee").insert({1, "james", 100000, 43429342});
StorageManager().getTable("Employee").findTuplesWithAttributeValue(id, 4);
StorageManager().getTable("Employee").deleteTuplesWithAttributeValue(name, "james
    ");
```

The storage manager interface would be as follows;

```
struct Table;
class StorageManager {
```

```
    map<string, Table> catalogue;

    public:
        Table& getTable(string name) {return catalogue[name]; };
};
```

In order to store tuples, we need to decide where and how we store data. Generally, for where we store it, we mainly consider disk or memory. The way we store data can vary from being sorted, compressed, RLE-encoded, etc. A simple table interface could be as follows;

```
struct Table {
    using AttributeValue = variant<int, float, string>;
    using Tuple = vector<AttributeValue>;

    void insert(Tuple) {};
    vector<Tuple> findTuplesWithAttributeValue(int attributePosition,
       AttributeValue value) {};
    void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue
       value) {};
};
```

There is a fundamental mismatch in storing tuples; relations are two-dimensional, whereas memory is one-dimensional, therefore tuples need to be linearized in one of the two mainstream strategies;

- **The N-ary Storage Model** (NSM)

  In this model, the entries added are stored one after the other, in a one-to-one mapping onto memory. This is also referred to as a **row store**. An example of this is as follows;

  ```
  struct NSMTable : public Table {

      // not the tuple exposed to the outside
      struct InternalTuple {
          Tuple actualTuple;
          bool deleted = false;
          InternalTuple(Tuple t) : actualTuple(t) {};
      }
      vector<InternalTuple> data;

      void insert(Tuple);
      vector<Tuple> findTuplesWithAttributeValue(int attributePosition,
         AttributeValue value) {};
      void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue
          value) {};
  };

  // declared as member of a class (just append to a vector)
  void NSMTable::insert(Tuple t) { data.push_back(t); }

  vector<Table::Tuple> NSMTable::findTuplesWithAttributeValue(int
     attributePosition, AttributeValue value) {
      vector<Tuple> result;
      for (size_t i = 0; i < data.size(); i++) {
          if (data[i].actualTuple[attributePosition] == value && !data[i].
             deleted) {
              result.push_back(data[i].actualTuple);
          }
  ```

```
    }
    return result;
}

// we don't want to move everything therefore we just mark them as deleted
void NSMTable::deleteTuplesWithAttributeValue(int attributePosition,
    AttributeValue value) {
    for (size_t i = 0; i < data.size(); i++) {
        if (data[i].actualTuple[attributePosition] == value) {
            data[i].deleted = true;
        }
    }
}
```

However, this may not always be a good idea, as operations will require us going over the entire vector. Recall that we are assuming everything is in memory, however locality still matters (since the data may be further apart depending on the size of the tuple).

Memory is organised in **cache lines** (usually 64 bytes in size). When a core needs something, it asks L1, L2, and L3 cache first, before going to memory, and retrieves the entire cache line that the data resides on (and cache into L1 cache). If we then access something on the same cache line, then the access is almost free (in terms of time). More tuples will therefore fit on a single cache line if they are smaller. Cache lines can also be referred to as **blocks** or **pages**.

N-ary storage works well in inserting a new tuple, when we are inserting a tuple onto the same cache line. It also works well when we want to retrieve a tuple by its index. On the other hand, it's suboptimal when we want to check every row, which in the worst case will be across different cache lines.

- **Decomposed Storage Model** (DSM)

  On the other hand, in this model, entries are stored with the columns one after the other. This is also referred to as **column store**. An example of this implementation is as follows;

```
struct DSMTable : public Table {
    using Column = vector<AttributeValue>;
    vector<Column> data;
    vector<bool> deleteMarkers;

    void insert(Tuple);
    vector<Tuple> findTuplesWithAttributeValue(int attributePosition,
        AttributeValue value) {};
    void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue
        value) {};
};

// inserts cause tuple decomposition
void DSMTable::insert(Tuple tuple) {
    for (int i = 0; i < tuple.size(); i++) {
        data[i].push_back(tuple[i]);
    }
}

// find requires tuple reconstruction
vector<Table::Tuple> DSMTable::findTuplesWithAttributeValue(int
    attributePosition, AttributeValue value) {
```

```
    vector<Tuple> result;
    for (size_t i = 0; i < data[attributePosition].size(); i++) {
        if (data[attributePosition][i] == value && !deleteMarkers[i]) {
            Tuple reconstructedTuple;
            for (int column = 0; column < data.size(); column++) {
                reconstructedTuple.push_back(data[column][i]);
            }
            result.push_back(reconstructedTuple);
        }
    }
    return result;
}

void DSMTable::deleteTuplesWithAttributeValue(int attributePosition,
    AttributeValue value) {
    for (size_t i = 0; i < data.size(); i++) {
        if (data[attributePosition][i] == value) {
            deleteMarkers[i] = true;
        }
    }
}
```

Decomposed storage works well in the case when we are iterating over one column of a tuple. However, if these entries are one after the other, there is perfect data locality, which minimises the number of cache lines we need. On the other hand, insertion is suboptimal as we need to spread a tuple over memory. Similarly, accessing a single tuple, even when we know where it is in memory is suboptimal, as the tuple will need to be reconstructed.

- Hybrid Delta / Main Storage

```
struct HybridTable : public Table {
    DSMTable main; // every tuple will eventually end up here
    NSMTable delta; // will be inserted here then merged into main

    void insert(Tuple);
    vector<Tuple> findTuplesWithAttributeValue(int attributePosition,
       AttributeValue value) {};
    void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue
        value) {};
    void merge();
};

void HybridTable::insert(Tuple t) {
    delta.insert(t);
}

vector<Table::Tuple> HybridTable::findTuplesWithAttributeValue(int
    attributePosition, AttributeValue value) {
    vector<Tuple> results = main.findTuplesWithAttributeValue(
       attributePosition, value);
    vector<Tuple> fromDelta = delta.findTuplesWithAttributeValue(
       attributePosition, value);
    results.insert(results.end(), fromDelta.begin(), fromDelta.end());
    return results;
}
```

```
21
22 void HybridTable::deleteTuplesWithAttributeValue(int attributePosition,
       AttributeValue value) {
23     main.deleteTuplesWithAttributeValue(attributePosition, value);
24     delta.deleteTuplesWithAttributeValue(attributePosition, value);
25 }
26
27 void HybridTable::merge() {
28     for (auto i = 0u; i < delta.data.size(); i++) {
29         // these two operations need to atomic (otherwise we can return the
               same tuple multiple times)
30         main.insert(delta.data[i].actualTuple);
31         delta.data[i].deleted = true;
32     }
33 }
```

In conclusion, **DSM** works well fro scan-heavy queries (accessing a lot of tuples but few columns). This is common in analytical processing, and analytics mostly operate on historical data. On the other hand **NSM** works well for lookups and inserts. This is more common in transactional processing (inserting sales item, looking up products, etc). Transactions mostly operate on recent data. Hybrid exploits the aforementioned workloads, but it needs regular migrations (`merge()`) which may need to lock the database.

Catalogue

The catalogue stores metadata. The idea is that real-life data follows patterns, which if recognised and exploited, can lead to more efficiency. However metadata will need to be stored and maintained. Examples of metadata can be type, min / max values (if data is requested outside a range we know it doesn't exist), histograms, etc. Two simple ones are **sortedness** and **denseness**. Consider the following example, which is a common pattern (as the first column is often IDs);

```
1  class Table {
2      vector<Tuple> storage;
3      bool firstColumnIsSorted = true;
4      bool firstColumnIsDense = true;
5
6      public:
7          void insert(Tuple t) {
8              if (storage.size() > 0) {
9                  firstColumnIsSorted &= (t[0] >= storage.back()[0]);
10                 firstColumnIsDense &= (t[0] == storage.back()[0] + 1);
11             }
12             storage.push_back(t);
13         }
14
15         vector<Tuple> findTuplesWithAttributeValue(int attribute, AttributeValue
               value) {
16             if (attribute == 0 && firstColumnIsDense) {
17                 return { data[value - storage.front()[0]] };
18             } else if (attribute == 0 && firstColumnIsSorted) {
19                 if (binary_search(data, attribute, value)[0] == value) {
20                     return { binary_search(data, attribute, value) };
21                 }
22             }
```

```
23              ... // same scan as before
24          }
25
26          // we can also exploit the fact that order of rows doesn't matter, and
                therefore we can reorganise
27          void analyse() {
28              // this sort can be expensive (and therefore should be run regularly
                    but not always)
29              sort(storage.begin(), storage.end(), [](auto l, auto r) { return l[0]
                    < r[0] });
30              firstColumnIsSorted = true;
31              firstColumnIsDense = true;
32              for (size_t = 1; i < storage.size(); i++) {
33                  firstColumnIsDense &= storage[i][0] == storage[i - 1][0] + 1;
34              }
35          }
36  }
```

Variable Length Data

It's important to note that strings tend to have different lengths. However, we'd like to maintain fixed tuple sizes, as it allows for random access to tuples by their position. We can either overallocate space for `varchars` (for example, some system require a size parameter for maximum length), or we can store them out of place. Overallocating leads to strings under the maximum length being padded with some sort of terminator. This is good for locality and is simple to implement, however it's very wasteful for space (especially if they are too generous with lengths).

On the other hand, out of place storage now contain an index into a dictionary, which contains the string (all null terminated). However, retrieving a value will need an access to obtain the index and another to get the actual value (which has poor locality). While this is better for space (and is what most programming languages do), it's also complicated (and causes difficult garbage-collection). This gives an optimisation (**dictionary compression**) however - if we find a string that has already been used when we insert, we can use the address of the existing value for the insert.

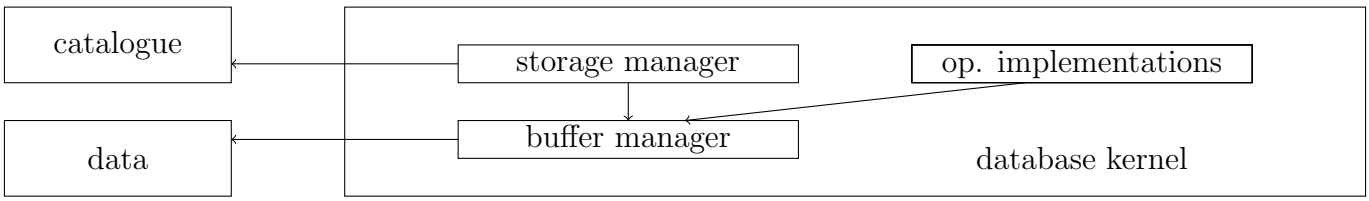
Data Storage on Disk

Disks are different from main memory in a number of ways;

- **larger pages** kilobytes compared to bytes
- **higher latency** milliseconds compared to nanoseconds
- **lower throughput** hundreds of megabytes instead of tens of gigabytes per second

  this is why we consider DBMSs as I/O bound
- **OS gets in the way** filesize can be limited, therefore DMBS need to map files and offsets

Our goals also change slightly; disks now dominate cost, and therefore complicated I/O management strategies can pay off. Due to larger pages, each page behaves like a mini-database in the case of N-ary storage. The basic kernel now looks like the following (where the catalogue is in memory, but the data is in disk);



The **buffer manager** manages disk-resident data. It maps unstructured files (large 'blobs of bytes') to structured tables for reading and writing. Since DBs don't trust OSs, it makes sure files have a fixed size, and it also safely writes data to disk when necessary. It also writes in **open (for writing) pages**. An example implementation is as follows;

```
class BufferManager;
class Table {
    BufferManager& bufferManager;
    string relationName = "Employee";

    public
        void insert(Tuple t) {
            bufferManager.getOpenPageForRelation(relationName).push_back(t);
            bufferManager.commitOpenPageForRelation(relationName);
        }

        vector<Tuple> findTuplesWithAttributeValue(int attribute, AttributeValue
           value) {
            vector<Tuple> result;
            auto pages = bufferManager.getPagesForRelation(relationName);
            for (size_t i = 0; i < pages.size(); i++) {
                auto page = pages[i];
                for (size_t i = 0; i < page.size(); i++) {
                    if (page[i][attribute] == value) {
                        result.push_back(page[i]);
                    }
                }
            }
            return result;
        }
}

struct BufferManager {
    using Tuple = vector<AttributeValue>;
    using Page = vector<Tuple>;

    size_t tupleSize;
    map<string, vector<Tuple>> openPages;
    map<string, vector<string>> pagesOnDisk; // maps one relation to many pages on
        disk (filenames)
    size_t numberOfTuplesPerPage();

    vector<Tuple>& getOpenPageForRelation(string relationName);
    void commitOpenPageForRelation(string relationName);
    vector<Page> getPagesForRelation(string, relationName);
}

vector<Tuple>& BufferManager::getOpenPageForRelation(string relationName) {
    return openPages[relationName]; // creates one if needed
}

void BufferManager::commitOpenPageForRelation(string relationName) {
    while (openPages[relationName].size() >= numberOfTuplesPerPage(tupleSize)) {
        vector<Tuple> newPage;
```

```
48
49          // move overflowing tuples to new page
50          while (openPages[relationName].size() > numberOfTuplesPerPage(tupleSize))
                {
51              newPage.push_back(openPages[relationName].back());
52              openPages[relationName].pop_back();
53          }
54
55          pagesOnDisk[relationName].push_back(writeToDisk(openPages[relationName]));
56          openPages[relationName] = newPage; // contains tuples that didn't fit on
                disk
57      }
58  }
59
60  vector<vector<Tuple>> BufferManager::getPagesForRelation(string relationName) {
61      vector<vector<Tuple>> result = { openPages[relationName] };
62      for (size_t i = 0; i < pagesOnDisk[relationName].size(); i++) {
63          result.push_back(readFromDisk(pagesOnDisk[relationName][i]));
64      }
65      return result;
66  }
```

However, we still need to determine `numberOfTuplesPerPage`;

- **unspanned pages** pages like mini-databases

  The goal of this is simplicity, and good random access performance; we want to find the record with a single page lookup, given a `tuple_id`. When we don't have enough space on a page for another tuple, we will write to disk (even when not full) and obtain a new page. However, we can quite easily infer which page a tuple will be on, since we know how many tuples are on each page. This cannot deal with large records (where the size is larger than a page), nor can we have in-page random access if the records are variable size.

  ```
  1  const long pageSizeInBytes = 4096;
  2  size_t BufferManager::numberOfTuplesPerPage() {
  3      return floor(pageSizeInBytes / tupleSizeInBytes);
  4  }
  5
  6  // space consumption of a relation of n tuples;
  7  ceil(data.size() / mnumberOfTuplesPerPage())
  ```

- **spanned pages** optimising for space efficiency

  On the other hand, the goal here is to minimise space waste and to support large records. Spanned pages can have tuples existing across pages - this is complicated and also hurts random access performance, since we can no longer easily determine where a tuple is. We can calculate the number of pages per relation as follows;

  ```
  1  long dataSizeInBytes = tupleSizeInBytes * data.size();
  2  long numberOfPagesForTable = ceil(dataSizeInBytes / pageSizeInBytes);
  ```

  However - we cannot determine the number of tuples per page as this is not constant.

- **slotted pages** random access for in-place NSM

  This is the most complicated method covered, but is also what is used by most systems. This stores tuples in in-place N-ary format, and stores the tuple count in the **page header** (at the start of the page). Offsets are also stored to every tuple, which are filled in from the **end** of the

page (hence the index of the first tuple is the last item in the page and so on). Offsets need to be typed large enough to address page;

- `byte`s for pages smaller than 256 bytes
- `short`s for pages smaller than 65,536 bytes
- `int`s for pages smaller than 4 gigabytes

Some disk-based database systems also keep a dictionary per page, which solves the problem of variable sized records (and allows for duplicate elimination, at the granularity of a single page, as previously mentioned). A global dictionary is not used as it will lead to more accesses.