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){};
};
```

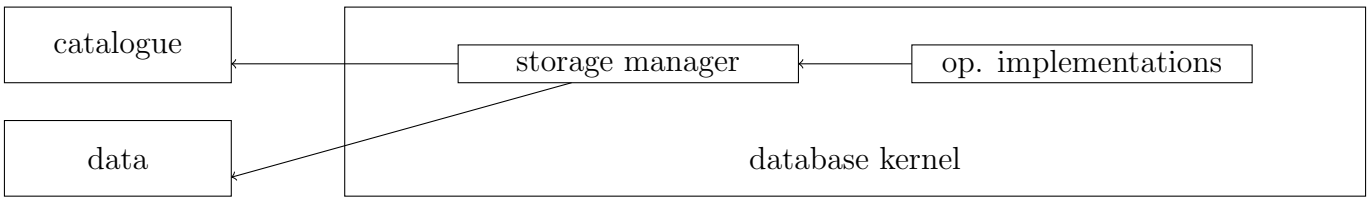
Since relational algebra is closed, operators can be combined as long as signatures are respected (cross product takes two inputs, whereas selections and projections take one);

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

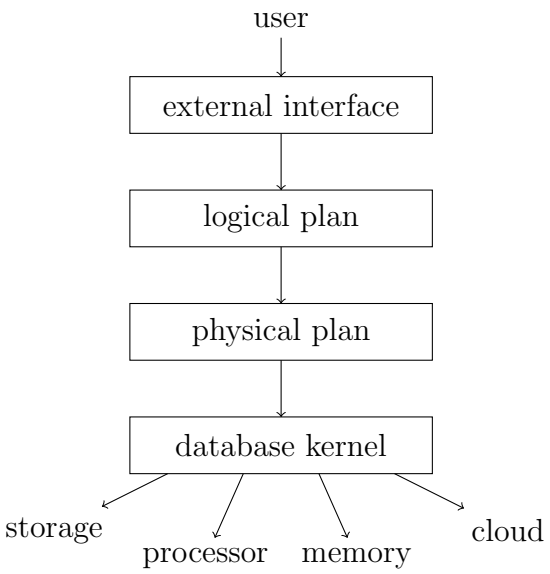
Lecture 2

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.

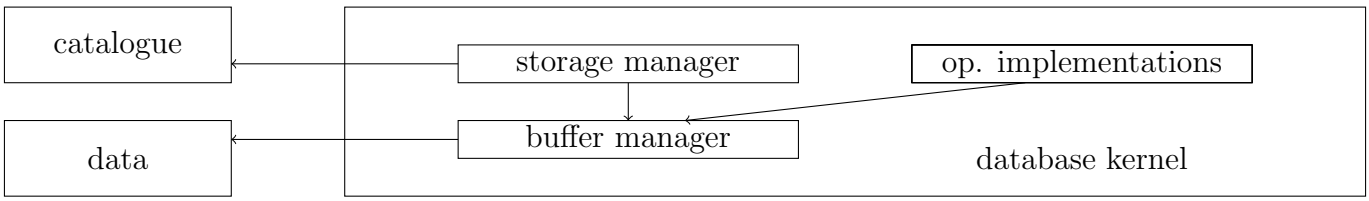
Data Storage on Disk

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.

Lecture 3

Join

The issue with data normalisation (see **CO130**) is that data ends up scattered across different tables. For example, consider the following example tables. Notice how a simple application is now split across these 4 tables - but we care more about who ordered what book, rather than the order IDs.

Customer

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

Order

ID	CustomerID
1	1
2	2
3	3

OrderedItem

ID	CustomerID
1	1
1	2
2	1
3	3

Book

ID	Title	Author
1	Fahrenheit 451	Ray Bradbury
2	Animal Farm	George Orwell
3	Distributed Systems	van Steen & Tanenbaum

Joins are very common not only due to normalisation (with mostly **Foreign-Key joins**), but also due to the value produced by combining data. For example, we can find users that purchase the same product with joins, or finding what advertisements work (seeing which users search for a term within a timeframe after seeing an advert).

Joins are basically cross products with a selection **involving both inputs**. The variations on joins are based on how tuples without corresponding rows on either input are matched;

- **left join** $(R \overset{\text{L}}{\bowtie} S)$

  This returns all rows in R, even if no rows in S match (in this case it fills the columns of S with `NULL` values)

- **right join** $(R \overset{\text{R}}{\bowtie} S)$

  Same as above, but the inverse

- **full outer join** $(R \overset{\text{O}}{\bowtie} S)$

  Returns every row in R as well as every row in S (even if no rows are matching) - similarly will fill with `NULL` values if nothing is matching;

$$R \overset{\text{O}}{\bowtie} S \equiv (R \overset{\text{L}}{\bowtie} S) \cup (R \overset{\text{R}}{\bowtie} S)$$

An example of the matching function is as follows;

```
SELECT * FROM R JOIN S ON (R.r = S.s)
```

The matching function for joins does not have to be equality;

- **equi-join** (algorithmically equivalent to intersections) equality
- **inequality joins** inequality constraint (< or >)
- **anti-join** <> or !=
- **Theta joins** all other joins (difficult to optimise)

Join Algorithms

Some of the join algorithms are as follows;

- **nested loop join**

```
using Table = vector<vector<int>>;
Table left, right;
for (size_t i = 0; i < leftRelationSize; i++) {
    auto leftInput = left[i];
    for (size_t j = 0; j < rightRelationSize; j++) {
        auto rightInput = right[j];
        if (leftInput[leftAttribute] == rightInput[rightAttribute]) {
            writeToOutput({leftInput, rightInput});
        }
    }
}
```

This is a simple algorithm and is trivial to parallelise (since there are no dependent loop iterations). This also has sequential I/O, which is good for the buffer manager. The effort required however is;

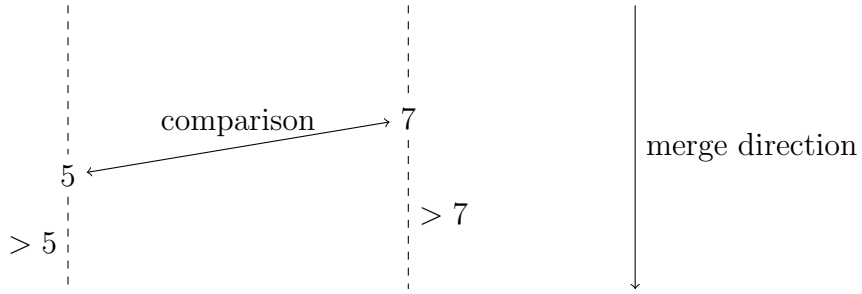
$$\Theta(|L| \times |R|) \text{ , reduced to } \Theta\left(\frac{|L| \times |R|}{2}\right) \text{ if uniqueness can be assumed}$$

- **sort-merge joins**

```
// this assumes values are unique and sorted
auto leftI = 0;
auto rightI = 0;
// keep iterating until either input is finished
while (leftI < leftInputSize && rightI < rightInputSize) {
    auto leftInput = left[leftI];
    auto rightInput = right[leftI];
    if (leftInput[leftAttribute] < rightInput[rightAttribute]) {
        leftI++;
    } else if (rightInput[rightAttribute] > leftInput[leftAttribute]) {
        rightI++;
    } else {
        writeToOutput({leftInput, rightInput});
        rightI++;
        leftI++;
    }
}
```

The idea of this is that we first sort both inputs and have cursors starting at the start of each input. If the value at the left pointer is smaller than the value at the right pointer, we increment the left pointer (and vice versa). If it's the same, then we can output a match.

Assuming (without loss of generality) the value of on the left is less than the value on the right. All values after the value on the right are greater than it (due to the sorted nature). Therefore no value after the value on the right can be a join partner to the value on the left, therefore we can advance the cursor on the left.



The effort required for this is as follows (the merge must go through each tuple on either side);

$$O(\underbrace{|L| \times \log |L|}_{O(\texttt{sort}(L))} + \underbrace{|R| + \log |R|}_{O(\texttt{sort}(R))} + \underbrace{|L| + |R|}_{O(\texttt{merge})})$$

This has sequential I/O in the merge phase, however it is tricky to parallelise. This also works for inequality joins, but we need to be more careful when advancing the cursors.

- **hash joins**

```
vector<optional<vector<int>>> hashTable; // optional since slots can be empty
int hash(int);
int nextSlot(int);

for (size_t i = 0; i < buildSide.size(); i++) {
    auto buildInput = buildSide[i];
    auto hashValue = hash(buildInput[buildAttribute]);

    // avoid overwriting a value
    while (hashTable[hashValue].hasValue) {
        hashValue = nextSlot(hashValue);
    }
    hashTable[hashValue] = buildInput;
}

for (size_t i = 0; i < probeSide.size(); i++) {
    auto probeInput = probeSide[i];
    auto hashValue = hash(probeInput[probeAttribute]);
    while (hashTable[hashValue].hasValue && hashTable[hashValue].value[
       buildAttribute] != probeInput[probeAttribute]) {
        hashValue = nextSlot(hashValue);
    }
    if (hashTable[hashValue].value[buildAttribute] == probeInput[
       probeAttribute]) {
        writeToOutput({hashTable[hashValue].value, probeInput})
    }
}
```

We need to first distinguish the **build-side** (side buffered in the hashtable) and **probe-side** (side used to look up tuples in the hashtable).

For this, we need to establish some requirements on the hash function. The requirements is that it is pure (and has no state), and need to know the output domain (the range of generated values)

in order to allocate the size. It's also ideal to have a contiguous output domain (without holes) and a uniform distribution (where all values are equally likely for consistent performance). Some common examples are;

- **MD5**
- **Modulo-Division** simplest (but input skew leads to output skew)
- **MurmurHash** one of fastest, decent hash-functions
- **CRC32** also has input skew issue, but has hardware support

Furthermore, we need to handle conflicts on hash collision. This needs to have some locality (if there is too much locality, there can be issues when inserting a lot of data into the same slot). Additionally, it needs to have no holes (we want to probe all slots to avoid memory waste). Some approaches are as follows;

- **linear probing**

  When a slot is filled, try the next one and continue doing so until a free slot is found - wrapping around to the start at the end of the buffer. This approach is simple and has great access locality. However it can lead to long probe-chains for adversarial input data (if we have input locality).
- **quadratic probing**

  This is similar to the above strategy, but we instead double the distance from the initial slot each time, first checking one that is a distance of 1 away, then 2, then 4, and so on, also wrapping at the end of the buffer. This is also simple, but only has good locality for the first few probes (and gets much worse). The first few probes are also likely to cause conflicts.
- **rehashing**

  We want to distribute probes uniformly, which can cause poor access locality but reduces conflicts. To do so, we can just use the hash function again. However, to ensure all slots are probed, we need to consider **cyclic groups**.

An example of this applied, with **modulo hashing** (with a constant factor of 10 for simplicity) and **linear probing** is as follows;

```
1 int hash(int v) { return v % 10; }
2 int probe(int v) { return (v + 1) % 10; }
3
4 buildSide = {1, 2, 7, 8, 12, 16, 17};
```

With this, we'd end up with the following slots in the hash table (- denotes unused);

```
(index 0) => [-, 1, 2, 12, -, -, 16, 7, 8, 17] <= (index 9)
```

Hash joins give us sequential I/O on inputs but pseudo-random access to the hash-table during build and probe. It's parallelisable over the values on the probe side, but parallelising the build is difficult. The effort required is;

$$\Theta(|\texttt{build}| + |\texttt{probe}|) \text{ (best case) and } O(|\texttt{build}| \times |\texttt{probe}|) \text{ (worst case)}$$

However, it's also important to consider that we typically want to store more than a single value, and often we may not have the uniqueness guarantee. Good hashing is still expensive and requires lots of CPU cycles (more expensive than multiple data accesses). Slots are often also allocated in buckets, allow for more than one tuple per slot. It's roughly equivalent to rounding every hash value down to a multiple of the bucket size. Sometimes buckets are implemented as linked lists (**bucket-chaining**, **open addressing**) which is a bad idea for lookup performance. Hashtables are also arrays, which occupy space (typically overallocated by a factor of two to reduce long

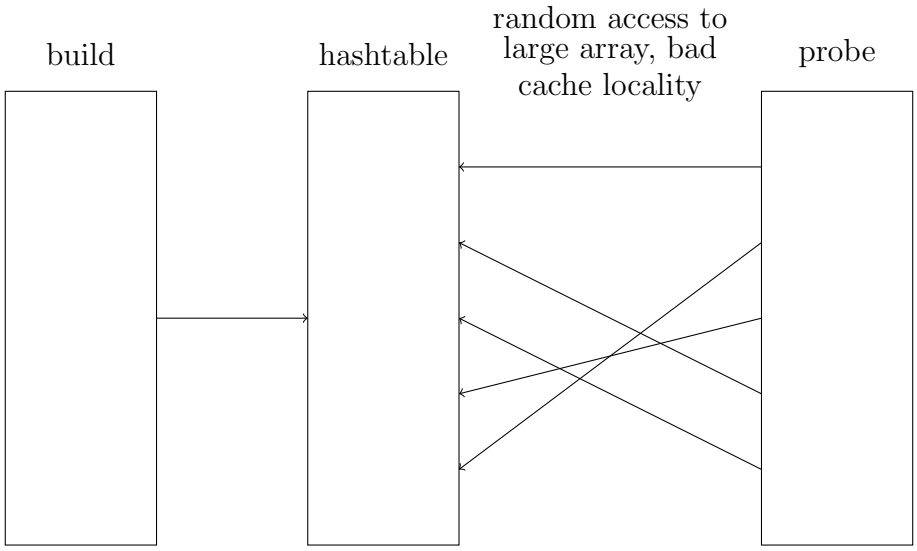
probe chains). They are also probed randomly in the probe phase, ideally we want to keep the hash table in memory / cache, but not to disk. **For this class**, if the hashtable doesn't fit, **every access has a constant penalty**. Generally we use hash joins when one relation is much smaller than the other.

Partitioning

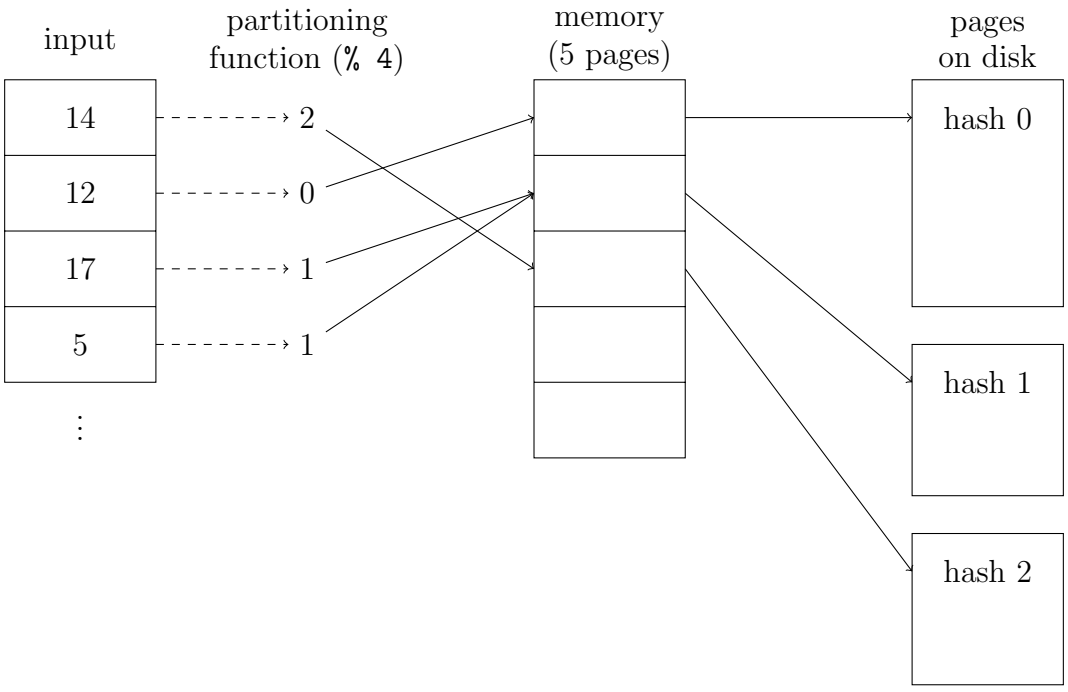
We have the fundamental premise that sequential access is much cheaper than random access, and the difference grows with the page size. If we assume a random value access cost c, the sequential value access cost is;

$$\frac{c}{\text{pagesize}_{\text{OS}}}$$

Assuming the hashtable doesn't fit in the page / cache, we will use the cost c quite frequently, and therefore the extra pass for partitioning, which is mostly sequential access might be worth doing. We can visualise thrashing as follows;



On the other hand, by using a partitioning function (which typically has a small range to fit stuff into memory);



The write from input to memory is done for every tuple, but the write from memory to the pages on the disk is only done when the page is **filled**, which doesn't waste bandwidth (and also lowers the

number of cs we use). As such, the probing becomes more involved, we first partition the probe in the same way, and we then only look at the partitioned hashtables, which gives us localised random access.

Another benefit of partitioning is that we can now parallelise the processing of each of the smaller joins, since we know they are disjoint.

Indexing

We should note that these algorithms are done in phases (build / probe and sort / merge). Typically the first phase, build or sort, is independent of the query phase, and can be done before any data is requested (such as when it is first loaded in). This is called indexing.

An index is a secondary storage, which is about replicating data. The replication is controlled by the DBMS, which means they can be created and destroyed without breaking the system. These replicas are semantically invisible to the user (doesn't change results). On the other hand, these replicas occupy space and need to be maintained under updates.

A primary index is used to store the tuples of a table, not a table, and therefore only one of these can be used per table. On the other hand, a secondary index stores pointers to the tuple of a table, hence we can have many of these in a table.

In SQL, an index can be created and destroyed as follows (however, it's unclear what type of index is created, nor do we have control over parameters);

```
1 CREATE INDEX index_name ON table_name (column1, column2, ...);
2 DROP INDEX index_name;
```

Some examples are as follows;

- **Hash-Indexing**

  The first step of a hash-join was building a hash-table, instead the hash-index is the same but persistent. In an unclustered hash-index, there is hash table which stores the key and position, and in the relation it is ordered by the position (and contains the actual data).

  The **ephemeral** hashtables we built for hash-joins were temporary, and we assume that no new tuples are added during the evaluation of the query, and we'd also know a rough amount of tuples that would end up in the table. However, all of this changes if the hash-table is persistent. Since persistent tables may grow arbitrarily large, we need to overallocate by a lot. If the fill-factor grows beyond x percent, we need to rebuild it with a larger overallocation - this can be expensive (and leads to load spikes on the inserts causing rebuilds). Similar, deletion can also be problematic.

  Previously, we used empty slots to mark an end of a probe-chain. However, a value must remain if something is deleted - we can either leave the value and mark it as deleted, or put the last value in the probe chain in the position. A deletion strategy (to delete key `k`) is as follows;

  - hash `k`, find `k`, and keep a pointer to `k`
  - continue probing until the end of the probe chain is found
  - if the value at the end of the chain has the same hash as `k`, move it into `k`'s slot
  - otherwise mark `k` as deleted (we can then fill this slot with the next value that hashes into the probe chain)

  An example is as follows. Note that 14 has the same hash as 23 (under modulo 9), but 14 has a different hash to 17.

before				
key	non-key			deleted
9	16	9	34	
27	5	61	45	
12	12	78	1	
23	84	17	69	
5	45	71	20	
17	9	42	83	
14	21	55	2	

after delete 23				
key	non-key			deleted
9	16	9	34	
27	5	61	45	
12	12	78	1	
14	21	55	2	
5	45	71	20	
17	9	42	83	

after delete 14				
key	non-key			deleted
9	16	9	34	
27	5	61	45	
12	12	78	1	
14	21	55	2	×
5	45	71	20	
17	9	42	83	

Similar to hash-joins, persistent hash-tables are good for hash-joins and aggregations (assuming they are built on the join / aggregation key columns). They also reduce the number of candidates for equality selections, but don't help on much else.

- **Bitmap-Index**

  It's first important to establish a **bitvector** is a sequence of 1-bit values indicating a boolean condition holding for the elements of a sequence of values. However, it's important to note that CPUs don't work with individual bits but rather words (let us assume 8 bit words, although in reality it's at least 32 bits);

$$\begin{aligned} BV_{==7}([4,7,11,7,7,11,4,7]) &= [0,1,0,1,1,0,0,1] \\ &= 128\cdot 0 + 64\cdot 1 + 32\cdot 0 + 16\cdot 1 + 8\cdot 1 + 4\cdot 0 + 2\cdot 0 + 1\cdot 1 \\ &= 89 \end{aligned}$$

  Bitmap indices are a collection of bitvectors on a column (one for each distinct value in that column). This is useful if there are few distinct values and these bitvectors are disjoint.

column	bitmap index		
	==5	==3	==9
5	1	0	0
3	0	1	0
9	0	0	1
9	0	0	1
9	0	0	1
5	1	0	0
3	0	1	0
9	0	0	1

  An implementation of this is as follows;

```
1  unsigned char** bitmaps;
2  void scanBitmap(byte* column, size_t inputSize, byte value) {
3
4      // finds specific bitmap for a certain value
5      unsigned char* scannedBitmap = bitmapForValue(bitmaps, value);
6
7      // iterate over bytes in bitmap
8      for (size_t i = 0; i < inputSize / 8; i++) {
9          // skip if none of the values are set
10         if (scannedBitmap[i] != 0) {
11             unsigned char bitmapMask = 128; // binary 10000000
12             for (size_t j = 0; j < 8; j++) {
```

```
13                  if ((bitmapMask & scannedBitmap[i]) && column[i * 8 + j] ==
                        value) {
14                      writeOutput(column[i * 8 + j]);
15                  }
16                  bitmapMask >>= 1;
17              }
18          }
19      }
20  }
```

This is useful as it reduces the bandwidth need for scanning a column, in the order of the size of the type of the column in bits (instead of loading a 32 bit value, we can just check 1 bit). Predicates can now also be combined using logical operators on the bitvectors. Additionally, arbitrary boolean conditions (such as ranges) can be indexed by some systems.

Binned bitmaps uses the idea of having n bitvectors, each with a predicate covering a different part of the value domain. For example if we assume our column type is a `byte`, we can have the following three bins;

- bin 1: 0 - 7
- bin 2: 8 - 20
- bin 3: 21 - 255

We need to ensure the conditions span the entire value domain. Also, the index cannot distinguish values in a bin (unless it only contains one value) - it can only produce **candidates**, some of which may be false positives.

Generally there are two strategies for binning;

- **equi-width**

  This is simple to configure - we can calculate our bin-width to be

  $$\frac{\max(\texttt{column}) - \min(\texttt{column})}{\texttt{numberOfBins}}$$

  This has limited use when indexing non-uniformly distributed data (since there can be a high frequency of false positives in a highly populated bin).
- **equi-height**

  This method is more resilient against non-uniformly distributed data - the false positive rate is value independent. We want to have the same number of counts per bin. The construction is more difficult however, as we need to determine quantiles and is usually performed on samples. If the distribution changes, it will need to be rebinned (which can be expensive).

Additionally, when we are binning, we may encounter many consecutive values which are equal. Each of these consecutive values can be replaced with the value (the Run) as well as the number of tuples (length) - this works very well on high-locality data. For example, we can transform `[0, 1, 1, 1, 1, 1, 0, 1]` to `[(0, 1), (1, 5), (0, 1), (1, 1)]`. However, this will now require a sequential scan, which can be fixed with **length prefix summing** (which stores the start of the run instead of the length); `[(0, 3), (1, 3), (0, 1), (1, 1)]` to `[(0, 0), (1, 3), (0, 6), (1, 7)]`. If we were to search for a value, we can just do a binary search instead of a sequential scan.

- **B-Trees**

  Databases are typically I/O bound (for disk), therefore we want to minimise page I/O operations. There are also many equality lookups, as well as many updates. Databases use high-fanout

trees to minimise page I/Os. We want the node of a tree to ideally be exactly the same size as a page (if we access a page, we might as well get the maximum amount of information out). Generally, a node in a B-Tree contains pairs (pivots) of keys and payloads, as well as child pointers to other nodes between the pairs (as well as before and after).

A B-Tree is defined as follows;

- a **balanced tree** with out-degree n (every node has $n-1$ keys)
- the **root** has at least one element
- each **non-root node** contains at least $\lfloor \frac{n-1}{2} \rfloor$ key / value pairs (at least half full)

Unlike other trees, B-Trees grow towards the root. In order to maintain balance, on an insertion the following steps apply;

- find the correct **leaf-node** to insert by walking the tree and inserting the value
- if this node overflows (the leaf was already full), split the node in two halves
  - take one pivot from the middle of the leaf and move it to the parent node
  - the two newly created nodes will become the left and right children of the pivot
- if the parent overflows, repeat the procedure on the parent
- if the parent is root, a new root node is introduced (therefore we add more roots, rather than add leaves)

For deletion, the following steps apply;

- find the value to delete
  - delete it if it is in a leaf node
  - otherwise, if it is in an internal node, keep a pointer to it, and replace it with the maximum leaf-node value from the left-child (removing the value from the leaf-node) - all modifications are on the leaf level
- if the affected leaf node underflows, we need to rebalance the tree bottom-up
  - try to obtain an element from a neighbouring node (to the right), move that up one level and make it the new splitting pivot, and take the old splitting pivot from the parent and put it into the leaf node
  - however, if that fails (the neighbour isn't more than half full), the nodes can be merged and the parent splitting key can be removed
  - if that causes an underflow, keep rebalancing upwards

While B-Trees can support ranges, it is complicated and requires going up and down the tree. This causes many node traversals, however node sizes are usually the same as page sizes, therefore traversals translate into page faults (which we want to minimise). Leaf pointers also aren't used, and most of the data lives in leaf nodes, therefore there is some space wasted.

- **B$^+$-Trees**

  The idea of this is to make range scans faster by keeping data only in the leaves, and linking leaf nodes to the next. Inner-node split values are replicas of leaf-node values. There is now more of a distinction between inner nodes and leaf nodes. The child pointers of the inner nodes are to other nodes, whereas the child pointers of leaf nodes (to the left) are now the payloads for a given key, and the right-most child pointer points to the next leaf. As such, the leaf nodes now form a sorted sequence.

  The balancing is largely the same as the balancing for regular B-Trees, however the deletion of an inner-node's split values imply a replacement with a new value from the leaf node.

- **Foreign-Key Index**

  In SQL, a foreign key is as follows; `ALTER TABLE Orders ADD FOREIGN KEY (BookID_index) REFERENCES Book(ID)`. FK constraints specify that for for every value that occurs in an attribute of a table, there is exactly one value in the PK column of another table. This constraint must be done by the DBMS, on an insertion or update, the DBMS needs to look up the PK value, and instead of storing the value, the DBMS could store a pointer to the referenced PK or tuple. We can think of this as equivalent to a pointer, and we can consider these as pre-calculated joins (which is often used since most joins are PK/FK joins from normalisation).

  Generally there are very few downsides - they cause insignificant work under updates, they do not cost much more space, and doesn't take more effort for query optimisation.

Lecture 4

Processing Model

A processing model is the mechanism used to connect different operators. This matters since different storage models are optimised for different bottlenecks, such as data access, CPU, query compilation, etc.

Function Objects

These can be referred to in different ways, such as lambdas, function pointers, etc. On a higher level, they are pieces of code that are treated like data (basically just pointers to an instruction) - they can be assigned to variables, invoked with arguments, and will return a value.

```
Python: aggregate = lambda v, t : v + t[0]
C++:    function<int(int, Tuple)> aggregate = [](int v, Tuple t) {
          return v + (int)t[0];
        };
```

Volcano Processing

This influential system was built in the 80s, and focused strongly on design practices, with less of a focus on performance. It contained components such as the cascades query optimiser, a non-relational physical algebra, and a query processing model. This was designed with flexibility, clean design, maintainability, developer productivity, and a fair bit of tunability in mind.

The operator interface provided is as follows - they are connected using pointers (in our case the `unique_ptr` smart pointer);

```
struct Operator {
  virtual void open() = 0;
  virtual optional<Tuple> next() = 0;
  virtual void close() = 0;
}
```

Looking at the operators;

- **scan** read a table and return contained tuples one by one

  ```
  struct Scan : Operator {
    Table input;
    size_t nextTupleIndex = 0; // operators are stateful
    Scan(Table input) : input(input){};
    void open(){}; // no work to be done at the start
    optional<Tuple> next() {
  ```

```
    return nextTupleIndex < input.size()
                ? input[nextTupleIndex++]
                : {}; // empty value marks end
  };
  void close(){};
};
```

- **projection** transform one tuple into another using a projection function

```
struct Project : Operator {
  using Projection = function<Tuple(Tuple)>;
  Projection projection;
  unique_ptr<Operator> child;
  void open() { child->open(); };
  optional<Tuple> next() { return projection(child->next()); };
  void close() { child->close(); };
};
```

- **selection** return all tuples satisfying a boolean predicate

```
struct Select : Operator {
  using Predicate = function<bool(Tuple)>;
  unique_ptr<Operator> child;
  Predicate predicate;

  Select(unique_ptr<Operator> child, Predicate predicate)
      : child(move(child)), predicate(predicate) {};

  void open() { child->open(); };
  optional<Tuple> next() {
    // iterate until matches or cannot find a tuple
    for (auto candidate = child->next(); candidate.has_value(); candidate =
       child->next()) {
      if (predicate(candidate)) {
        return candidate;
      }
    }
    return {};
  };
  void close() { child->close(); };
};
```

- **union** return all tuples from one child followed by all tuples from another

```
struct Union : Operator {
  unique_ptr<Operator> leftChild;
  unique_ptr<Operator> rightChild;
  void open() {
    leftChild->open();
    rightChild->open();
  };
  optional<Tuple> next() {
    auto candidate = leftChild->next();
    return candidate.has_value() ? candidate : rightChild->next();
  };
  void close() {
```

```
    leftChild->close();
    rightChild->close();
  };
};
```

- **difference** return all tuples from the left that are not in the right (buffer right first)

```
struct Difference : Operator {
  unique_ptr<Operator> leftChild;
  unique_ptr<Operator> rightChild;
  vector<Tuple> bufferedRight;

  void open() {
    leftChild->open();
    rightChild->open();
    for (auto rightTuple = rightChild->next(); rightTuple.has_value();
        rightTuple = rightChild->next()) {
      bufferedRight.push_back(rightTuple);
    }
  };
  optional<Tuple> next() {
    for (auto candidate = leftChild->next(); candidate.has_value(); candidate
        = leftChild->next()) {
      if (find(bufferedRight.begin(), bufferedRight.end(), candidate) ==
          bufferedRight.end()) {
        return candidate;
      }
    }
    return {};
  };
  void close() {
    leftChild->close();
    rightChild->close();
  };
};
```

Note that this is an example of a **pipeline breaker**; an operator that produces the first output tuple after **all** input tuples from one of the sides have been processed.

- **cross product**

```
struct Cross : Operator {
  unique_ptr<Operator> leftChild;
  unique_ptr<Operator> rightChild;
  Tuple currentLeftTuple{};
  vector<Tuple> bufferedRightTuples;
  size_t currentBufferedRightOffset = 0;

  void open() {
    leftChild->open();
    rightChild->open();
    currentLeftTuple = leftChild->next();
  };
  optional<Tuple> next() {
    // some processing moved from open() phase to next() phase
    auto currentRightTuple = rightChild->next();
```

```
    if (currentRightTuple.has_value()) {
      bufferedRightTuples.push_back(currentRightTuple);
    }
    if (currentBufferedRightOffset == bufferedRightTuples.size()) {
      currentBufferedRightOffset = 0;
      currentLeftTuple = leftChild->next();
    }
    return currentLeftTuple.concat(bufferedRightTuples[
       currentBufferedRightOffset++]);
  };
  void close() {
    leftChild->close();
    rightChild->close();
  };
};
```

Since we are able to implement this without breaking the pipeline, the implementation of an operator determines whether an operator is a pipeline breaker. Some operators do not have pipelineable implementations.

- **grouped aggregation** group tuples that are equal and calculate per-group aggregate(s)

```
using SupportedDatatype = variant<int, float>;
using AggregationFunction = function<SupportedDatatype(SupportedDatatype,
    Tuple)>;

struct GroupBy : Operator {
  unique_ptr<Operator> child;
  vector<optional<Tuple>> hashTable;
  Projection getGroupKeys;
  vector<AggregationFunction> aggregateFunctions;

  size_t nextSlot(size_t value);
  size_t hashTuple(Tuple t);
  // simplified for single attribute groups
  void open() {
    child->open();

    auto inputTuple = child->next();
    while (inputTuple.has_value()) {
      auto slot = hashTuple(inputTuple[groupAttribute]);
      while (hashTable[slot].has_value() && inputTuple[groupAttribute] !=
         hashTable[slot][0]) {
        slot = nextSlot(slot);
      }

      // new entry created
      if (!hashTable[slot].has_value()) {
        hashTable[slot][0] = { inputTuple[groupAttribute] };
        hashTable[slot].resize(aggregateFunctions.size() + 1)
      }

      for (size_t j = 0; j < aggregateFunctions.size(); j++) {
        hashTable[slot].data[j + 1] = aggregateFunctions[j](hashTable[slot][j
            + 1], inputTuple);
```

```
      }
      inputTuple = child->next();
    }
  }

  int outputCursor = 0;
  // all the work is done in the open() phase, just iterate here
  optional<Tuple> next() {
    while (outputCursor < hashTable.size()) {
      auto slot = hashTable[outputCursor++];
      if (slot.has_value()) {
        return slot.value();
      }
    }
    return {};
  };

  void close() { child->close(); };
};
```

This can be used as follows, to execute a plan;

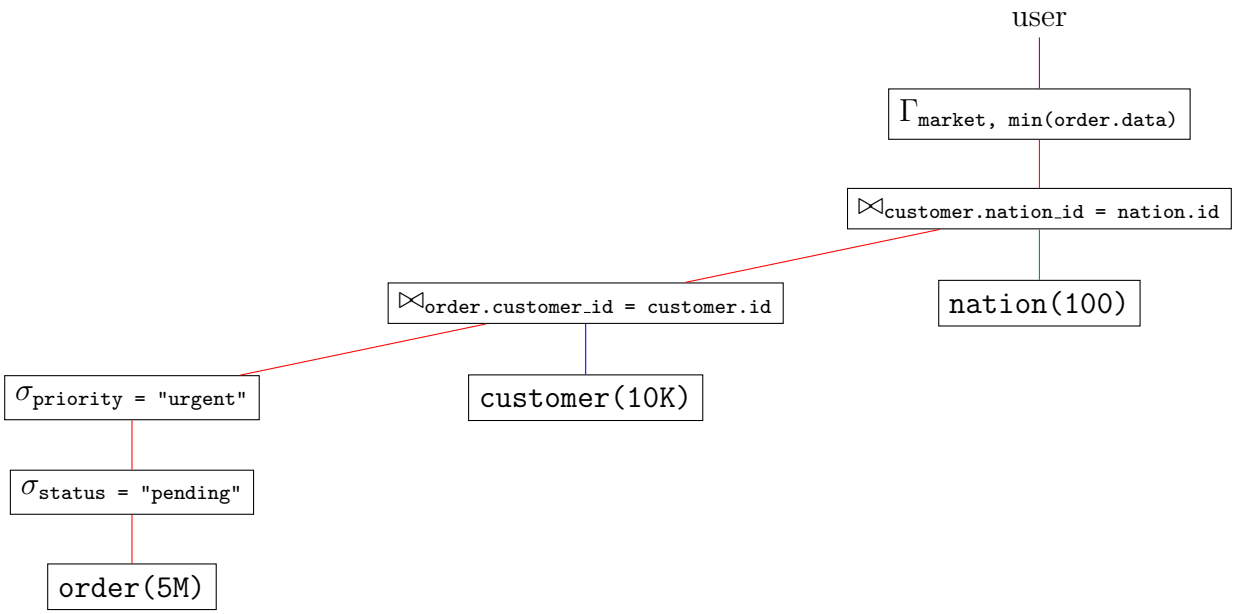
```
Table input{
  {1l, "James", "1 Main Street"},
  {2l, "Jacob", "4 Second Street"},
  {3l, "Peter", "1 Main Street"}
};

auto plan = make_unique<GroupBy>(
    make_unique<Select>(
      make_unique<Scan>(input),
      [](auto t) { return t[2] == string("1 Main Street"); }
    ),
    [](auto t) { return Tuple{t[1]}; },
    vector<AggregationFunction>{
      [](auto v, auto t) { return long(v) + 1 }
    }
);

plan->open();
for (auto t = plan->next(); t; t = plan->next()) {
    cout << t << endl;
}
```

Volcano has a number of advantages;

- short, roughly 200 lines for our implementation
- object-oriented design
- easily extensible (adhere to the interface)
- good I/O behaviour (tuples consumed as soon as they are produced)

Estimating Buffer I/O in Volcano

The following plan has the pipeline fragments highlighted (note that the build side is below the node);



Scans read all pages of the relation and can therefore be calculated in the same way as before (see storage lecture). However, in the case of a pipeline breaker, we have to consider the following cases;

- buffer and all other buffers in pipeline fragment fit in memory — no I/O
- otherwise
  - buffer accessed sequentially (cross product) — number of occupied pages (per pass)
  - buffer accessed randomly / out-of-order (hash aggregation) — one page access per tuple
    note that this is the worst case, when it isn't in memory

On the first point, we need to consider all the buffers in the pipeline fragment. For example, if we look at the red pipeline, there are three hash tables constructed, one for the aggregation, and one for each of the two joins, and we will need to consider all of them, since they must all live in memory at the same time.

We can also use the following rules to know whether a buffer fits in memory;

- buffers need to hold data according to their algorithm and input;
  - nested loop buffers and sorted relations take exactly their input (the number of tuples multiplied by the size of each tuple) - we also assume spanned pages
  - hashtables are overallocated by a factor (default to two if we aren't told)
- we also assume perfect knowledge about the input and output cardinalities
  - input buffer size is the input cardinality multiplied by the tuple size
  - output buffer size is the output cardinality multiplied by the tuple size
- we know the memory / buffer pool size (bytes or number pages)

Example of Buffer I/O Estimation

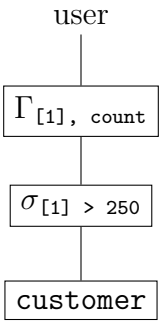
Consider the following scenario, where we have a selection selectivity of 30% (the number of tuples being selected), grouping cardinality of 9 (number of groups is 9), an overallocation factor of 2, a buffer pool of 512KB, and a page size of 64 bytes.

Our customer table has 10,000 tuples, with the following attributes;

- `id` — `int32`
- `name` — dictionary compressed (`int32` keys)

- `address` dictionary compressed (`int32` keys)
- `nation` `int32`
- `phone` `int32`
- `accountNumber` `int32`

Our pipeline is as follows;



From here we can do the following calculations;

$$\begin{aligned}
\texttt{CustomerTableSize} &= \texttt{no. attributes} \cdot \texttt{no. bytes} \cdot \texttt{no. tuples} \\
&= 6 \cdot 4 \cdot 10000 \\
&= 240000 \\
\texttt{CustomerTablePages} &= \lceil \frac{\texttt{CustomerTableSize}}{\texttt{bytes per page}} \rceil \\
&= \lceil \frac{240000}{64} \rceil \\
&= 3750 \\
\texttt{CustomerScanCosts} &= \texttt{CustomerTablePages} \\
\texttt{GroupingCardinality} &= 9 \\
\texttt{NumberOfAttributesInGroupingTable} &= 2
\end{aligned}$$

From here, we can work out the number size of the grouping hash table;

$$\texttt{GroupingHashTableSize} = \lceil 2 \cdot \texttt{GroupingCardinality} \cdot \texttt{NumberOfAttributesInGroupingTable} \cdot 4 \rceil$$

This gives us a result of 144 bytes, which is less than the buffer pool size and therefore can be ignored. Therefore, our total page I/O is just the customer scan costs (hence 3750). Note that the cost of selection is ignored, as it's not a pipeline breaker.

Problems with Volcano

However, there are some problems with Volcano;

- CPU efficiency

  Consider the cost of a sequential memory access. Assume a laptop's memory bandwidth is 37.5 GB/s, and has 4 cores at 2.9 GHz. This gives 9.375 GB/s per core, which comes to ≈ 3.23 bytes per cycle (round to one integer for simplicity). We therefore need to ensure we can process an integer per cycle for maximum efficiency.

- Function pointers

  The steps for the CPU (roughly) to evaluate a function call are as follows;

  - CPU stores current instruction pointer (the `call`)
  - arguments for the function are put on the execution stack

- CPU instruction pointer is set to the address of the first instruction of the function code (called a jump; `JMP`)
- function is executed until a return
- instruction pointer is set to instruction after `call`

The modern CPU executes instructions in stages; such as `fetch`, `decode`, `execute`, `memory read`, `write result`. For simplicity, assume instructions spend exactly one cycle in each stage (actually spends **at least** one in each stage). Let's also assume that it moves on to the next stage after every cycle.

Function pointers however cause pipeline bubbles (actually called control hazards). Since the jump instruction sets the instruction pointer to an arbitrary address, the next instruction can only be read after the jump is complete, since the CPU needs to read the next instruction from this address. This leads to the pipeline not being filled (hence not fully utilised). The cost of this is dependent on the number of stages in the CPU. The number of function calls can be counted as follows (per-tuple);

- **scan** none, tuples are read straight from buffer
- **selections / projections** one to read input, one to apply predicate
- **cross product inner / outer** one to read the input
- **join** one to read the input (inline hash function for HJ and comparison for SMJ)
- **group-by** one to read input, one to calculate each new aggregate value
- **output** one to extract for output

Working through the same example as before, we have the following;

$$
\begin{aligned}
\texttt{SelectFunctionCalls} &= \texttt{no. tuples} \cdot 2 && \text{read and apply} \\
&= 20000 \\
\texttt{GroupingFunctionCalls} &= \texttt{no. tuples} \cdot \texttt{Selectivity} \cdot 2 && \text{read and aggregate} \\
&= 12000 \\
\texttt{FinalOutputExtraction} &= 9 && \text{one per group} \\
\texttt{FunctionCalls} &= \text{total of above} \\
&= 26009
\end{aligned}
$$

Looking at the results above, assuming that a function call costs 15 cycles, and we can read one integer per cycle, we have the following;

metric	number	cycles per value	total cycles
function calls	26009	15	390135
accessed 64-byte pages	3750	16	60000

From these numbers, we can say that this is a CPU bound operation, which gets worse as the queries become more complex.

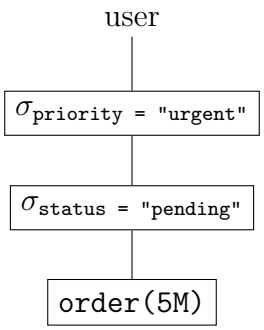
Bulk Processing

Since the CPU is the bottleneck, and function calls dominate the CPU cost, we want to process queries with as few function calls as possible. The idea is to turn **Control Dependencies** into **Data Dependencies**. Instead of processing tuples right away, we buffer them (fill the buffer with lots of tuples), and pass the buffer to the next operator.

For example, selection can be written as follows;

```
int select(Table& outputBuffer, Table const& input, int predicate, int
    attributeOffset) {
  for (size_t i = 0; i < input.size(); i++) {
    if (input[i][attributeOffset] == predicate) {
      outputBuffer.push_back(input[i]);
    }
  }
  return outputBuffer.size();
}
```

Consider a small section of the pipeline example;



The implementation, with the previously defined selection, would be as follows;

```
Table order, buffer1, buffer2;
int pendingCode = 5, urgentCode = 7;
select(buffer1, order, pendingCode, 1);
select(buffer2, buffer1, urgentCode, 2);
```

This causes a tight loop, with no function calls nor jumps. This is very CPU efficient, but every operator is now a pipeline breaker, and the calculation rules are similar - each operator reads and writes all of its inputs and outputs (respectively) sequentially, and the number of pages is calculated the same as before (in the storage lecture).

The calculations are quite similar to Volcano, however there's one difference in that all operators are cost-estimated completely independently, since everything is its own pipeline fragment.

Consider an example with a selectivity of 25%, 1 million tuples each with 9 32-bit attributes. Similarly, we have a 512 KB cache, and pages of 64 bytes.

$$
\begin{aligned}
\texttt{Selectivity} &= 0.25 \\
\texttt{CachelineSize} &= 64 \\
\texttt{BufferPoolCapacityInPages} &= \frac{512 \cdot 1024}{64} \\
\texttt{TableSizeInTuples} &= 1000000 \\
\texttt{TableTupleSize} &= 9 \cdot 4 \\
\texttt{TableSize} &= \texttt{TableSizeInTuples} \cdot \texttt{TableTupleSize} \\
\texttt{TablePages} &= \lceil \frac{\texttt{TableSize}}{\texttt{CachelineSize}} \rceil \\
\texttt{SelectionInputInPages} &= \texttt{TablePages} \\
\texttt{SelectionOutputInPages} &= \lceil \frac{\texttt{TableSize} \cdot \texttt{Selectivity}}{\texttt{CachelineSize}} \rceil \\
\texttt{SelectionIO} &= \texttt{SelectionInputInPages} + \texttt{SelectionOutputInPages} \\
&= 703125
\end{aligned}
$$

Bulk Processing and Decomposed Storage

The idea of **By-Reference Bulk Processing** is to save bandwidth by avoiding copying data. Instead of producing tuples, we produce IDs (32-bit positions in their buffers). When we process a tuple, we use the ID to look up actual values - this is basically a hashtable without conflicts.

```
int select(vector<int>& outputBuffer, optional<vector<int>> const&
    candidatePositions, int predicate, int attributeOffset, vector<Tuple> const&
    underlyingRelation) {
  // case where we are the first selection in the plan (all are candidates)
  if (!candidatePositions.has_value()) {
    for (size_t i = 0; i < underlyingRelation.size(); i++) {
      if (underlyingRelation[i][attributeOffset] == predicate) {
        outputBuffer.push_back(i);
      }
    }
  } else {
    for (size_t i = 0; i < candidatePositions->size(); i++) {
      if (underlyingRelation[(*candidatePositions)[i]][attributeOffset] ==
          predicate) {
        outputBuffer[outputCursor++] = (*candidatePositions)[i];
      }
    }
  }
  return outputCursor;
}
```

The implementation of our previous result would now be as follows (note that the code above might be *slightly* incorrect);

```
vector<Tuple> order;
vector<int> buffer1, buffer2;
int pendingCode = 5, urgentCode = 7;
auto buffer1Size = select(buffer1, {}, pendingCode, 1, order);
auto buffer2Size = select(buffer2, buffer1, urgentCode, 2, order);
```

The rules for estimating buffer I/O are slightly different; we now read all of its **candidate buffer** and write all of its candidate buffer sequentially, although the calculation remains the same. In addition, each operator resolves the candidate references by looking up the values in the base relation. The rules for temporary buffers are the same (again).

Page Access Probability

If we define the selectivity s as the percentage of tuples being touched, and n the number of tuples on a page, and also assume uniformly distributed values, we can define the probability of any one of them being touched as;

$$p(s, n) = 1 - (1 - s)^n$$

We can then estimate the number of page faults as;

$$p(\texttt{FirstSelectSelectivity}, \texttt{AverageBaseTableTuplesPerPage}) \cdot \texttt{BaseTablePages}$$

By-Reference Bulk Processing of Decomposed Data

We can save even more bandwidth. We know that every operator processes exactly one column of a tuple. As such, N-ary storage is quite wasteful (since values of a tuple are co-located on a page), therefore we will pay the cost to access all values on a page, even if we only process one (this will

also occupy space in the buffer pool). DSM fixes this; this was introduced as a consequence of bulk processing. Using the same example as before, with the selectivity of 30%;

$$\begin{aligned}
\texttt{SelectSelectivity} &= 0.3 \\
\texttt{CachelineSize} &= 64 \\
\texttt{BufferPoolCapacityInPages} &= \frac{512 \cdot 1024}{64} \\
\texttt{OrderTableSizeInTuples} &= 10000 \\
\texttt{OrderTupleAttributeSize} &= 1 \cdot 4 && \text{only one attribute}
\end{aligned}$$

We can then work out the sizes for the order table;

$$\begin{aligned}
\texttt{OrderTableColumnSize} &= \texttt{OrderTupleAttributeSize} \cdot \texttt{OrderTableSizeInTuples} \\
\texttt{OrderTableColumnPages} &= \lceil \frac{\texttt{OrderTableColumnSize}}{\texttt{CachelineSize}} \rceil
\end{aligned}$$

Now we can look at the sizes for the selection - note that the output is actually the candidate positions, however we don't need to even consider it in our calculation (last line);

$$\begin{aligned}
\texttt{SelectionInputInPages} &= \texttt{OrderTableColumnPages} \\
\texttt{SelectionOutputInTuples} &= \texttt{OrderTableSizeInTuples} \cdot \texttt{SelectSelectivity} \\
\texttt{SelectionOutputInPages} &= \lceil \frac{\texttt{SelectionOutputInTuples} \cdot 4}{\texttt{CachelineSize}} \rceil \\
\texttt{SelectionIO} &= \texttt{SelectionInputInPages}
\end{aligned}$$

For the grouping, we need to first calculate the hash table size (note that this combined with the selection output is still smaller than the buffer pool capacity, hence we can ignore it again);

$$\texttt{GroupingHashTableSize} = \lceil 2 \cdot \texttt{GroupingCardinality} \cdot \texttt{NumberOfAttributesInGroupingTable} \cdot 4 \rceil$$

$$\begin{aligned}
\texttt{OrderTuplesPerPage} &= \frac{\texttt{CachelineSize}}{\texttt{OrderTupleAttributeSize}} \\
\texttt{GroupingIO} &= \lceil p(\texttt{SelectSelectivity}, \texttt{OrderTuplesPerPage}) \cdot \texttt{OrderTableColumnPages} \rceil \\
&\quad + \lceil \frac{9 \cdot 2 \cdot 4}{\texttt{CachelineSize}} \rceil
\end{aligned}$$

Therefore our total IO is 1438.

Topic 0 - Introduction

A distributed database is several databases at different sites, connected by server racks, LAN, or the Internet. This adds considerations such as network bottlenecks and failures.

We can prove that there's no finite protocol where we can be certain that two sites agree on an update to a value. If we assume some finite chain, of n messages, then the final one is not needed (since that may be unreliable), hence we have $n - 1$, and so on until we can show there are no messages in this protocol.

CAP Theorem

This leads to the CAP theorem, where no distributed system can contain **all three** of the following properties;

- **consistency** — all nodes see the same version of data
- **availability** — the system always responds within a fixed upper limit of time

- **partition tolerance**

  system always gives correct responses even when messages are lost or network failures occur

We can have **CA** with a single system (a centralised database), **CP** with a traditional distributed RDBMS. The latter waits if the network becomes partitioned (until it recovers), therefore we lose availability. **AP** is available in some Internet protocols such as DNS, where we sacrifice consistency.

Approaches to Distributed Databases

There are generally two approaches to DDB;

- **Heterogeneous**

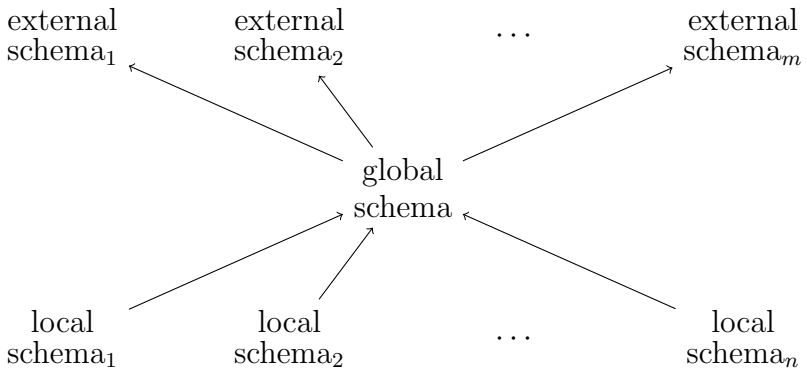
  These vary in database technologies (and are managed by different database administrators), and are designed at different times. Some example technologies are ODBC or JDBC.

  One consideration for this is that there may be different structures between the databases (for example RDBMS queried with SQL versus a CSV file queried by reading the file), and therefore a common data model (CDM) is required between them. Another consideration is a semantic difference, for example a table for bank branches may have different names for the fields, and may omit the prefix of a sort code - this requires **schema integration**.

  The diagram for the heterogeneous

- **Homogeneous**

  On the other hand, this is designed at the same time, and all use the same database technology. An example of this is an RDBMS cluster, or Map Reduce (e.g. Hadoop).

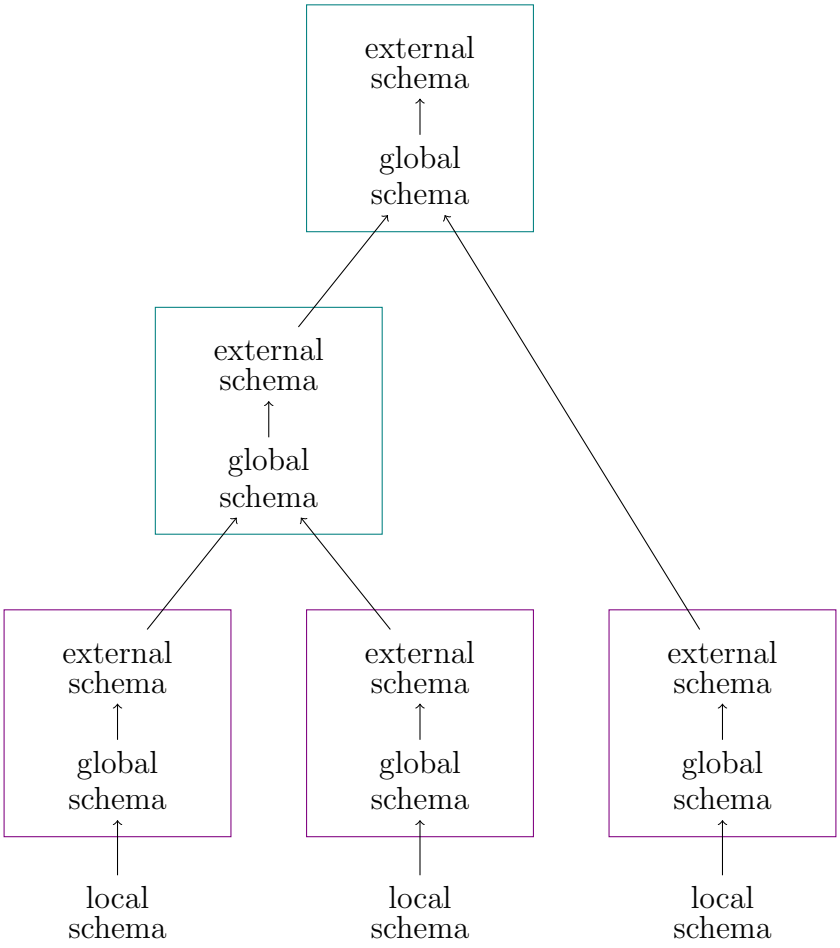


Operational Model

In the mediator architecture, we have the following, where virtual data is represented in teal and materialised data is represented in violet - note that the mediators are the two that are virtual data, and the three wrappers are the materialised data.

On the other hand, in the data warehouse approach, they are all materialised data (and the two that were previously teal are the data warehouses). Note that mediators request from the source, whereas database warehouses store the data from the source (hence materialised).

Both approaches have advantages and disadvantages, but it's important to note that the global schema in a data warehouse is essentially a cache and therefore needs to be updated. On the other hand, using virtual data increases the traffic and load since more queries have to be made to the local schemas (and wrappers).



Topic 1 - Data Distribution

A running example for this are the following tables;

- `branch`
  - `sortcode` (key)
  - `bname` (key)
  - `cash`
- `account`
  - `no` (key)
  - `type`
  - `cname`
  - `rate?`
  - `sortcode` (foreign key to `branch(sortcode)`)
- `movement`
  - `mid` (key)
  - `no` (foreign key to `account(no)`)
  - `amount`
  - `date`

We can briefly look at the types of data distribution as follows;

- **remote tables**

  Consider a global schema G with the tables R, T, U, V. This is split into sites S_1, S_2, S_3 with tables $(R), (T, U), (V)$ respectively. Therefore reads r and writes w must be distributed correspondingly. For example, a $r[R], w[R], r[T], r[U]$ on G becomes $r[R], w[R]$ on S_1, and $r[T], r[U]$ on S_2.

- **fragmentation**

  Consider a similar global schema with the same sites. The idea of fragmentation is to split objects $o_a, o_b, \ldots$ of a relation R between sites. The distribution is similar to before, but on an object level.

  There are a few approaches to this;

  - **horizontal fragmentation** $R_1 = \sigma_{P_1} R, \ldots, R_n = \sigma_{P_n} R$

    In this case, we split the table by rows, via a rule that **partitions** the data (prevents a row from being in multiple sites) and is roughly even. For example, the original `account` table can be split into the following;

    $$\texttt{account}_1 = \sigma_{\texttt{type == current}}\texttt{account}$$
    $$\texttt{account}_2 = \sigma_{\texttt{type != current}}\texttt{account}$$

    Compared to a single site version, the DDB is faster (since we can run multiple of the same query in parallel across the sites), but less reliable (since the chance of one of the sites failing during a long query is higher).

    The worst attribute to fragment on something that would change over time, in our example it would be `rate`.

  - **derived horizontal fragmentation** $R_1 = R \ltimes S_1, \ldots, R_n \ltimes S_n$

    For this, we need to look at the semi-join;

    $$R \ltimes T = R \bowtie \pi_{attr(R) \cap attr(T)} T$$

    This gives us the same columns as R, but only with the rows that have a value in T. For our running example, we might not have all accounts in the table $\texttt{account} \ltimes \texttt{movement}$, if it has nothing in `movement`.

    This allows us to fragment the `movement` table as follows, thus storing the movements at the same site as the respective accounts;

    $$\texttt{movement}_\texttt{i} = \texttt{movement} \ltimes \texttt{account}_\texttt{i}$$

  - **vertical fragmentation** $R_1 = \pi_{attrs_1} R, \ldots, R_n = \pi_{attrs_n} R$

    In this case we split by columns, simply with projections (however each attribute must be represented in at least one site, and there must be a key which we can join on, in this case `no`);

    $$\texttt{account}_1 = \pi_{\texttt{no, type, rate, sortcode}}\texttt{account}$$
    $$\texttt{account}_2 = \pi_{\texttt{no, cname}}\texttt{account}$$

  - **hybrid fragmentation**

    We can also combine the methods to fragment it further.

- **replication**

  Consider **objects** $o_a, o_b, \ldots$ on a global schema G. We can copy the data to all the sites, such that all sites $S_1, S_2, \ldots$ have a full copy of all objects. This allows queries to be run on **any** site, however updates will now need to write to **all** sites.

  A query on this is faster (in the parallel case) and is more reliable (due to site redundancy) than a single site. On the other hand, writes are slower but still more reliable. The writes need to be coordinated, thus causing it to be slower. However, the reliability of this depends on the implementation - if we do a synchronisation and one site goes down, the entire query fails. On the other hand, if we are able to recover from failure (and allow the rest of the query to continue after a site has been marked down), it can be more reliable with a recovery mechanism.

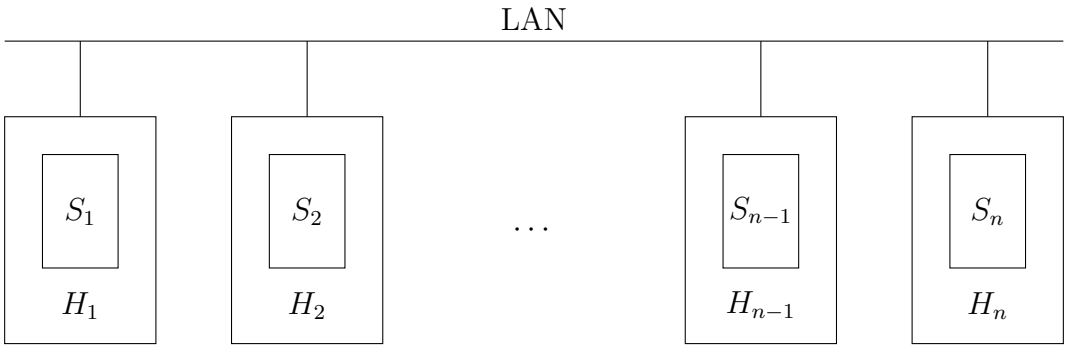
- **migration**

Topic 2 - Big Data

The traditional approach to dealing with larger amounts of data was to scale up (with larger, more powerful computers). However, this becomes increasingly more expensive, and will often require specialist hardware. The more modern approach is to scale out, which is to increase the number of commodity computers to spread the data.

Big Data System

A big data system is able to handle more data than fits on a single commodity computer and data that is spread over hundreds or thousands of servers. It must also handle the failures of nodes without any loss of data. As a consequence of the CAP theorem, availability is prioritised over consistency.



Data Models and MapReduce

Before we look at MapReduce, we should consider the different data models;

- **Key-Value**

  In this model data is stored as key-value pairs. This is schema-less, and has very limited querying capabilities (but useful for implementing cache). Examples of this are *Memcache* or *Redis*.

- **Document**

  While this has some structure to it, such as JSON, it's still schema-less as we aren't forced to a single structure. This supports some queries for searching fields within a document, but cannot perform queries such as joins. As such, MapReduce is used for OLAP. Some examples of this are *CouchDB* or *MongoDB*.

- **Wide Column**

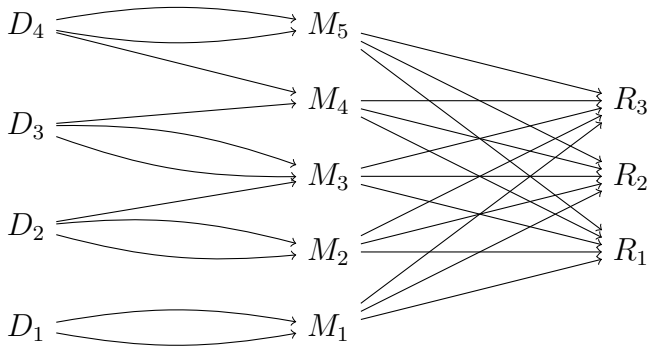
  Table data model with easy addition of new columns, and columns are put into families (allowing for vertical fragmentation on families). Also schema-less, and supports queries searching field values. MapReduce is also used for OLAP.

- **Relational**

  Schema based, and supports queries for searching fields and joins. Has ACID properties.

- **Graph** schema-less

The general structure of MapReduce is as follows;



Between the data nodes and the map nodes is the load phase, and between the map nodes and the reduce nodes is the shuffle phase. A hash function is used by the map nodes to decide which reduce nodes to to pass the data into in the shuffle phase (or some sort of rule that is known by all the map nodes).

We might notice that some of the work can be done in the map nodes (for example counting words), and this introduces the **combine** operation, which is a partial reduction, this reduces the inefficiency between the map nodes and the reduce nodes.

In the word count example, we have the following actions;

- map — obtain all words as a pair with a 1 (count)
- combine — partially reduce to give a word with a frequency
- reduce — fully reduce with all map nodes

MapReduce was originally reported by Google, and an open source implementation with the Hadoop family is as follows;

- Hadoop Distributed File System (HDFS)

  Standard storage by Hadoop applications, fragmentation by default in 128MB blocks, and defaults to three nodes of replication.

- Hadoop — Java implementation of MapReduce
- Hive — SQL language translated into Hadoop applications
- Pig Latin — high level scripting language to write Hadoop applications
- HBase — Java API to provide big table, provides scalable read / write access

Topic 3 - Pig Latin (RA)

Accessing Data

The `LOAD` operator reads in a TSV (by default) file as a relation.

```
account =
    LOAD '/path/to/account.tsv'
    AS (no:int, type:chararray, cname:chararray, rate:float, sortcode:int);
```

Relational Algebra

Some of the basic components of the relational algebra are implemented in Pig Latin as follows;

- project (π) $\pi_{\texttt{sortcode}}$ - `FOREACH <ALIAS> GENERATE <COLNAME>, ...`

```
account_sortcode_bag =
    FOREACH account
    GENERATE sortcode;

account_sortcode =
    DISTINCT account_sortcode_bag; # set
```

- select (σ) $\sigma_{\texttt{rate>0}}$ - `FILTER <ALIAS> BY <PREDICATE>`

```
account_with_rate =
    FILTER account
    BY rate > 0.0;
```

- product ($\times$) `CROSS <ALIAS> <ALIAS>`

```
branch_account_with_rate =
    CROSS branch, account_with_rate;
```

- join ($\bowtie$) - only performs equi-joins `JOIN <ALIAS> BY <COLNAME>, <ALIAS> BY <COLNAME>`

```
branch_with_interest_account =
    JOIN branch by branch::sortcode,
         account_with_rate BY account_with_rate::sortcode;
```

- union ($\cup$) $\pi_{\texttt{sortcode}}\texttt{branch} \cup \pi_{\texttt{no}}\texttt{account}$ - `UNION <ALIAS>, <ALIAS>`

```
branch_sortcode =
    FOREACH branch
    GENERATE sortcode;
account_no =
    FOREACH account
    GENERATE no;
all_ids_bag =
    UNION branch_sortcode, account_no;
all_ids =
    DISTINCT all_ids_bag;
```

- difference ($-$)

  Note that here there is no direct implementation. It has be to done by performing a left join and then keeping rows with null values. For example, performing the following;

$$\pi_{\texttt{no}}\texttt{account} - \pi_{\texttt{no}}\texttt{movement}$$

```
account_and_movement =
    JOIN account BY no LEFT,
    movement BY no;
account_without_movement =
    FILTER account_and_movement
    BY movement::no IS NULL;
account_no_without_movement =
    FOREACH account_without_movement
    GENERATE no;
```

Topic 4 - Pig Latin (Aggregation)

For this, we need to look at `GROUP` and `FLATTEN` (which reverts a `GROUP`). Note that when a group is made, it creates two columns `group` (which contains the attribute key we're grouping on), and `<ALIAS>`, which contains a bag of tuples (from the original relation).

```
accounts_movement =
    GROUP movement
    BY no;

movement_copy =
    FOREACH account_movements
    GENERATE FLATTEN(movement)

account_balance =
    FOREACH account_movements
    GENERATE group AS no, # rename
             SUM(movement.amount) AS balance; # accesses bag structure
```

The aggregation operators available to us are as follows;

function	result
`int COUNT(bag)`	returns the number of **not null** values in the bag
`int COUNT_STAR(bag)`	returns the number of values in the bag (**including null**)
`double AVG(bag)`	returns the average of the values in the bag
`double MAX(bag)`	returns the maximum value in the bag
`double MIN(bag)`	returns the minimum value in the bag
`double SUM(bag)`	returns the sum of the values in the bag
`bag DIFF(bag a, bag b)`	returns tuples in `a` that do not appear in `b`

To achieve something similar to SQL, we need to use the `GROUP` operator, and then apply the aggregation operators to the bag of tuples.

To optimise the previous query, we can reduce the amount of elements we store (if we aren't using them);

```
movement_data =
    FOREACH movement,
    GENERATE no, amount;

accounts_movement =
    GROUP movement_data
    BY no;

account_balance =
    FOREACH account_movements
    GENERATE group AS no,
             SUM(movement.amount) AS balance;
```

The above will give the same result, but the `accounts_movement` relation will be much smaller, we will only preserve the `amount` field.

Nested Statements

The following is a statement in SQL;

```
SELECT   account.no
         COUNT(movement.mid) AS no_trans,
```

```
3           SUM(CASE WHEN amount > 0.0 THEN amount ELSE 0.0 END) AS credit,
4           SUM(CASE WHEN amount < 0.0 THEN amount ELSE 0.0 END) AS debit,
5   FROM    account LEFT JOIN movement ON account.no=movement.no
6   GROUP BY account.no
```

Converted to Pig Script, we have the following;

```
1   account_and_movement =
2       JOIN account BY no LEFT,
3            movement BY no;
4   account_detail =
5       GROUP account_and_movement BY account::no;
6   account_credits_and_debits =
7       FOREACH account_detail {
8           credit =
9               FILTER account_and_movement
10              BY     amount > 0.0;
11          debit =
12              FILTER account_and_movement
13              BY     amount < 0.0;
14          GENERATE group AS no,
15                   COUNT(account_and_movement) AS no_trans,
16                   SUM(credit.amount) AS credit,
17                   SUM(debit.amount) AS debit;
18      }
```

WHERE and HAVING

Consider the following example in SQL;

```
1   SELECT account.cname,
2          SUM(movement.amount) AS balance
3   FROM account
4        JOIN movement ON account.no=movement.no
5   WHERE ABS(movement.amount)>100
6   GROUP BY account.cname
7   HAVING SUM(movement.amount)>200
```

In Pig, the same query would be done as follows;

```
1   account_movement_join =
2       JOIN account BY no, movement BY no;
3
4   account_movement_large =
5       FILTER account_movement_join
6       BY ABS(amount) > 100;
7
8   # optimisation
9   account_movement =
10      FOREACH account_movement_large
11      GENERATE cname, amount;
12
13  customer_details =
14      GROUP account_movement BY account::cname;
15
16  customer_balance_all =
```

```
17      FOREACH customer_details
18      GENERATE group AS cname, SUM(account_movement.movement::amount) AS balance;
19
20  customer_balance_large =
21      FILTER customer_balance_all
22      BY balance > 200
```

Lecture 5 - Pig Latin (MapReduce)

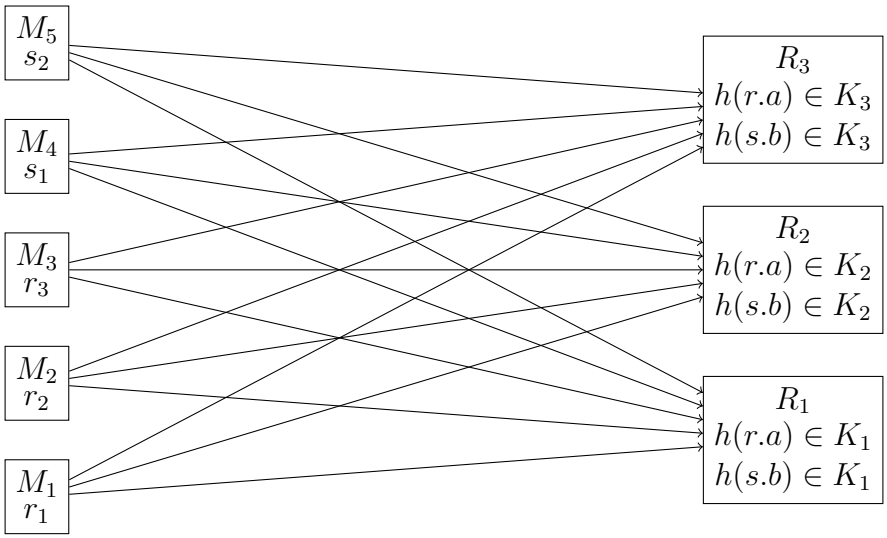
Pig scripts are interpreted into a sequence of Hadoop Map, Combine, Shuffle, and Reduce operations. Typically, Map and Combine processes are run on the nodes which contain data. The number of Reduce nodes is specified in Pig script, and defaults to 1. Temporary files are used to allow output of one MapReduce process to be fed into another. The only optimisation that is done by the Pig interpreter is to automatically push projections (from `GENERATE` in Pig) inside joins, anything else should be done by the script author. The translations are as follows;

Pig operator	map / reduce
`FILTER` R `BY` $A == val$	map
`FOREACH` R `GENERATE` $A, B, \ldots$	map
`CROSS` R, S	reduce
`GROUP` R `BY` A	combine (if possible), reduce
`JOIN` R `BY` A, S `BY` B	reduce
`JOIN` R `BY` A `LEFT OUTER`, S `BY` B	reduce
`JOIN` R `BY` A `RIGHT OUTER`, S `BY` B	reduce
`UNION` R, S	reduce

The number of reduce nodes can be controlled by a `PARALLEL` option at the end of the reduce operator.

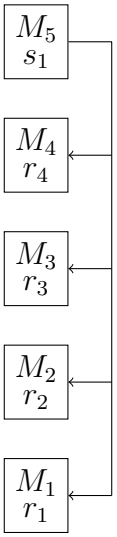
Distributed Hash Join

The standard `JOIN` will hash the value and send it to the correct hash bucket, note that for three reduce nodes, there are three corresponding buckets. The following diagram represents `JOIN r BY a, s BY b` with three reduce nodes.



Replicated Join

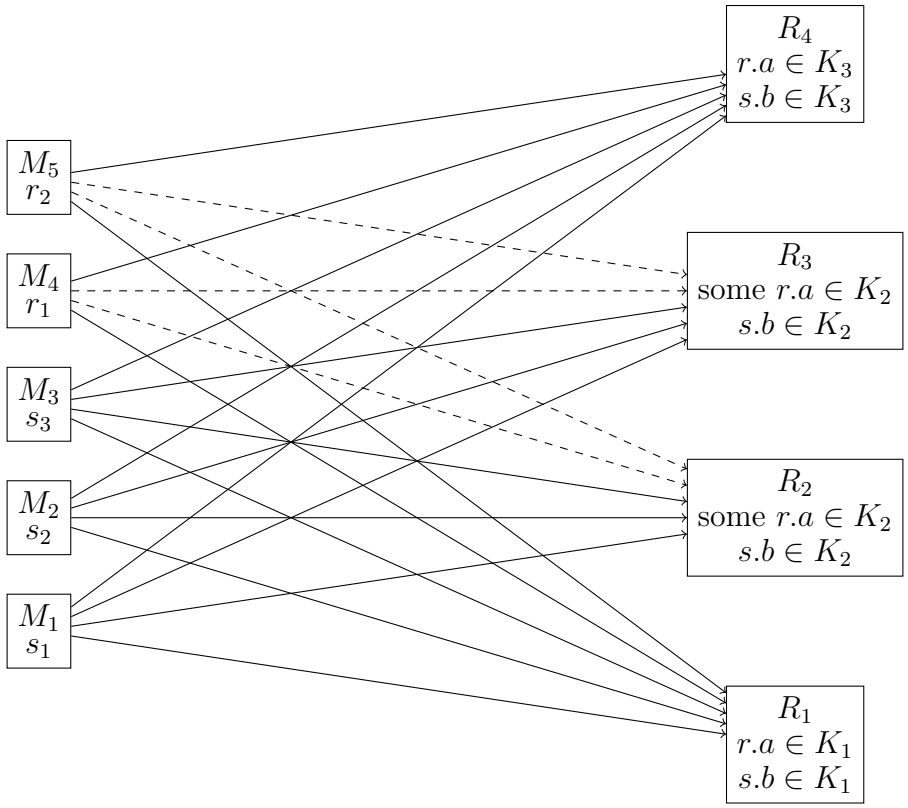
In a replicated join, the entire right hand table can be copied to all the map nodes holding the left hand table. These replicated joins are then executed as a Map process. The following diagram represents `JOIN r BY a, s BY b USING 'replicated'`, where `s` only resides on one map node;



Note that a `RIGHT` join cannot be done, since we cannot determine whether it joins or not on a particular node. For example, if we had a row in table s that doesn't join on anything; each of the map nodes can be certain it doesn't join, however it doesn't know if it joins on another node.

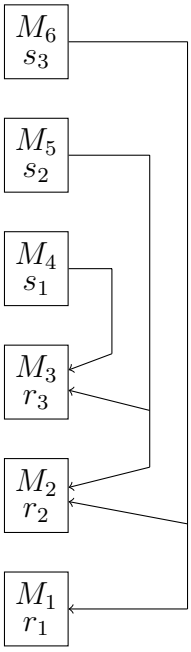
Skewed Join

In a skewed join, a histogram of the various join keys in r is first generated. If a key has a high frequency in r, the rows of r are distributed in a round robin fashion, and the rows of s are duplicated. The following diagram represents `JOIN r BY a, s BY b USING 'skewed'`;



Merge Join

This is a variation of SMJ, where it's assumed both inputs are already sorted. The first record of each block of s are sampled to determine the layout. The map nodes of r load s blocks as required. The following diagram represents `JOIN r BY a, s BY b USING 'merge'`;



Topic 6 - Distributing Queries

A traditional DBMS consists of the following three components;

- transaction manager

  Translates queries from a higher level language to a set of primitive operations, possibly optimising them in the process.

- scheduler

  Decides how to execute the aforementioned sequence of tasks concurrently to ensure the highest throughput.

- data manager

  Ensures that data is sent and read from disk in the most efficient way, and maintains durability of transactions.

In a distributed system, each site runs a local transaction manager (LTM), scheduler and data manager. There can be one or more sites with a global transaction manager, which dispatches the queries (and returns any data if required) based on predefined rules. The slowest part of distributed query processing is the communication between sites, and we'd like to minimise the amount of data sent between sites.

Global Transaction Manager Execution Plan

The GTM must transform transactions into sub-transactions for each site. If the table is fragmented, the transaction should be broken up into fragments (for example moving money between accounts, where the accounts reside on different databases can be done in two queries, one decrementing in the first table, another incrementing in the second). In the case of replicated tables, a read transaction only needs to go to one site, however any writing transactions will need to go to all sites.

In this example, assuming a horizontally fragmented relation;

```
1 -- from GTM
2 BEGIN TRANSACTION T4
3   SELECT SUM(cash)
4   FROM branch
5 COMMIT TRANSACTION T4
6
```

```
-- sent to LTM1 (sortcode<50)
BEGIN TRANSACTION T4_1
  SELECT @X=SUM(cash)
  FROM branch
COMMIT TRANSACTION T4_1

-- sent to LTM2 (sortcode>=50)
BEGIN TRANSACTION T4_2
  SELECT @Y=SUM(cash)
  FROM branch
COMMIT TRANSACTION T4_2

-- combined at GTM
BEGIN TRANSACTION T4
  SELECT @X + @Y
COMMIT TRANSACTION T4
```

RA Equivalences

We can use some equivalences in RA to improve our performance. Assume at S_1 we have $R_1(A, B, C)$ and $T_1(A, D, E)$, and at S_2 we have $R_2(A, B, C)$ and $T_2(A, D, E)$. At S_3, we want $\sigma_{A=n}R$. We can consider $R = R_1 \cup R_2$, hence the two are equivalent;

The cost of these can be analysed as follows (note that the lines in violet represent network transfers, and NDB stands for network data bytes);

$$NDB(P_n) = |R| \cdot \text{RowSize}(R)$$
$$NDB(P_o) = |\sigma_{A=n}R| \cdot \text{RowSize}(R)$$

If the attribute we are selecting over, in this case n, is part of the partition rule (such that we have $n \in K_1$, and $R_1 = \sigma_{A \in K_1}R$ and $R_2 = \sigma_{A \in K_2}R$), we can optimise even further by only considering R_1.

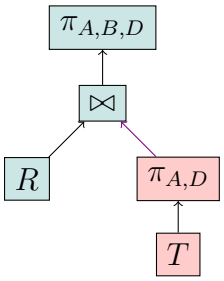
Similar logic can be applied in the case of vertical fragmentation; we can optimise a query if the columns we are considering are all in the same fragment (and therefore can ignore any other fragments).

Topic 7 - Distributed Joins

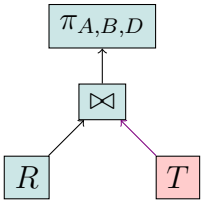
RA Equivalences Continued

Assume at S_1 we have $R(A, B, C)$, and at S_2 we have $T(A, D, E)$. We want to obtain $\pi_{A,B,D}(R \bowtie T)$ on S_1 (hence we only need stuff from S_2 to be sent over);

P_o



P_n

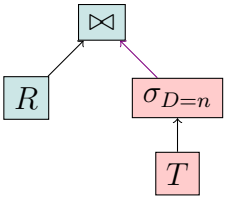


The cost of these can be analysed as follows;

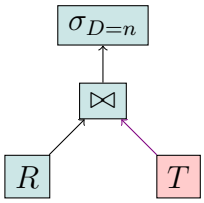
$$NDB(P_n) = |T| \cdot \text{RowSize}(T)$$
$$NDB(P_o) = |\pi_{A,D}T| \cdot \text{RowSize}(\pi_{A,D}T)$$

Using the same structure, assume we now want to perform $S_1 : \sigma_{D=n}(R \bowtie T)$;

P_o



P_n

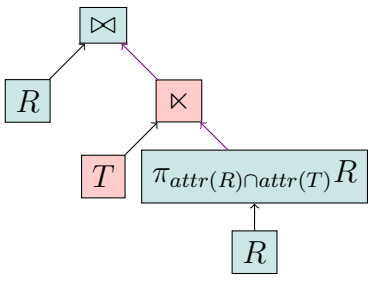


Similarly, we can notice that the optimised query sends less traffic (since we are reducing the number of rows we send);

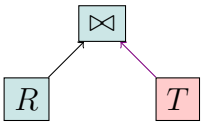
$$NDB(P_n) = |T| \cdot \text{RowSize}(T)$$
$$NDB(P_o) = |\sigma_{D=n}T| \cdot \text{RowSize}(T)$$

Another equivalence we can consider are direct and semi joins. For this scenario, we want to get $R \bowtie T$ at S_1. To do so, we can ensure that we only send over the rows of T that join with R, by first sending over the projection of T that has the joining attributes.

P_s



P_d



This gives us a resultant NDB as follows;

$$NDB(P_d) = |T| \cdot \text{RowSize}(T)$$
$$NDB(P_s) = |\pi_{attr(R)\cap attr(T)}R| \cdot \text{RowSize}(\pi_{attr(R)\cap attr(T)}R) + |T \ltimes R| \cdot \text{RowSize}(T)$$

It's important to note that this isn't always optimal, and therefore shouldn't always be applied (doing the first transfer can be wasteful).

For this example, where we consider **local and remote** joins, we have another relation in S_2, which is $U(E, F, G)$. Our goal is to obtain $S_1 : R \bowtie T \bowtie U$;

If we move the one of the joins onto the local server, we have the following;

$$NDB(P_l) = |T| \cdot \text{RowSize}(T) + |U| \cdot \text{RowSize}(U)$$
$$NDB(P_r) = |T \bowtie U| \cdot \text{RowSize}(T \bowtie U)$$

Similar to before, we cannot always be certain that this will offer an improvement.

Another equivalence to consider is a join over a derived horizontal fragmentation. Assume we wanted to obtain $S_3 : R \bowtie T$ and $T_n = T \ltimes R_n$, with $R_n(A, B, C)$ and $T_n(A, D, E)$ at S_1 and S_2. Instead of performing the join after taking the union ($R_1 \cup R_2$ and $T_1 \cup T_2$), the join could be done on each of the remotes;

This gives us the following query costs;

$$NDB(P_l) = |R| \cdot \text{RowSize}(R) + |T| \cdot \text{RowSize}(T)$$
$$NDB(P_r) = |R \bowtie T| \cdot \text{RowSize}(R \bowtie T)$$

Intersite Communication

If two relations are horizontally fragment, such that we have;

$$R = R_1 \cup \cdots \cup R_i, S = S_1 \cup \cdots \cup S_i$$

To perform a join on R and S will require intersite communication;

$$R \bowtie S = (R_1 \bowtie (S_1 \cup \cdots \cup S_i)) \cup \ldots (R_i \bowtie (S_1 \cup \cdots \cup S_i))$$

However, if derived horizontal fragmentation had been used, such that we have $R_i = R \ltimes S_i$, we require **no intersite communication**;

$$R \bowtie S \equiv (R_1 \bowtie S_1) \cup \cdots \cup (R_i \bowtie S_i)$$

As such, the design of a DDB schema should use derived horizontal fragmentation to avoid intersite communication for commonly used joins.

Topic 8 - Serialisability and Recoverability

This lecture starts with ACID properties of transactions, which I won't go over again.

Histories

Note that we will denote read operations on objects as $r_n[o_j]$ and writes as $w_n[o_j]$. At the end of the transaction, it will either result in c_n for commitment, or a_n if the transaction is aborted (undo all parts of the transaction). If b_n is at the start, it denotes the start of a transaction (not always given). For example;

```
1 BEGIN TRANSACTION
2   UPDATE branch
3   SET    cash = cash - 10000.00
4   WHERE  sortcode = 56
5
6   UPDATE branch
7   SET    cash = cash + 10000.00
8   WHERE  sortcode = 34
9 COMMIT TRANSACTION
```

Would be converted into the following ($H_1 =$)

- $r_1[b_{56}]$, `cash=94340.45`
- $w_1[b_{56}]$, `cash=84340.45`
- $r_1[b_{34}]$, `cash=8900.67`
- $w_1[b_{34}]$, `cash=18900.67`
- c_1

Note that the same pattern of transaction code will give the same pattern of operations;

```
1 BEGIN TRANSACTION
2   UPDATE branch
3   SET    cash = cash - 2000.00
4   WHERE  sortcode = 34
5
6   UPDATE branch
7   SET    cash = cash + 2000.00
8   WHERE  sortcode = 67
9 COMMIT TRANSACTION
```

Would be converted into the following ($H_2 =$)

- $r_2[b_{34}]$, `cash=18900.67`
- $w_2[b_{34}]$, `cash=16900.67`
- $r_2[b_{67}]$, `cash=34005.00`
- $w_2[b_{67}]$, `cash=36005.00`
- c_2

In a concurrent execution, the order of operations within each history is preserved, however interleaving can occur.

Serialisability means that a concurrent execution of transaction should have the same end result as a serial execution of the same transactions. **Recoverability** means that no transaction commits depending on data that has been produced by another transaction which hasn't yet committed. The latter can be clarified with an example; if the b_{34} read and write in H_2 is done first, before H_1, and then c_2 happens **before** c_1, we have something that is serialisable and recoverable.

Anomalies

We can have a number of anomalies occurring;

- **lost updates**

  If we were to have some sort of execution in the order $r_2[b_{34}], r_1[b_{34}], w_1[b_{34}], w_2[b_{34}]$, notice that the same value is read in for both, and therefore the first write ($w_1[b_{34}]$) will be lost. This is not serialisable, however it is recoverable since neither are reading a result that was changed by the other.

- **inconsistent analysis**

  This is the example at the start of the lecture, where we take the sum of the balances of the branches, which happens between a transaction (money is removed from one branch but **not** yet added to another). Note that the order of the commits at the end determine if it is recoverable (if c_1 commits first, then it is fine, since the sum transaction depends on the transfer).

- **dirty reads**

  A dirty read occurs when a transaction reads from data that another transaction has not yet committed. The example given for this is that $w_2[b_{34}]$ occurs before $r_1[b_{34}]$, and then c_1 occurs (after the write) and a_2 happens at the end, this is serialisable, but not recoverable.

- **dirty write**

  In this example, an updates are done on the same field on multiple accounts. If there is an interleaving between these two transactions, it results in some updates being applied from the first transaction and some being applied from the second.

- **phantom reads**

  The example given for this is an update on a rule (`rate < 5.5`), and then a new row gets inserted which would've satisfied that, which is then updated by another update in the first transaction.

- **write skew**

  This performs two different write operations based on overlapping information, where it could not have been performed if it were executed in a serial order.

We can summarise it as follows, where e_i means either c_i or a_i occurs, and $op_a \prec op_b$ means that op_a occurs before op_b in a history;

anomaly	pattern
dirty write	$w_1[o] \prec w_2[o] \prec e_1$
dirty read	$w_1[o] \prec r_2[o] \prec e_1$
inconsistent analysis	$r_1[o_a] \prec w_2[o_a], w_2[o_b] \prec r_1[o_b]$
lost update	$r_1[o] \prec w_2[o] \prec w_1[o]$

Serialisability

If we notice in some history (H) of anomalies a transaction that has been aborted, we can omit it and only look at the **committed projection** ($C(H)$).

A **conflict** occurs when there is an interaction between two transaction. The cases are when $r_x[o]$ and $w_y[o]$ are both in H, with $x \neq y$, or both $w_x[o]$ and $w_y[o]$ are in H, with $x \neq y$ again.

We define two histories H_i and H_j to be **conflict equivalent** ($H_i \equiv_{CE} H_j$) if they contain the same set of operations and order conflicts (of non-aborted transactions) in the same way. A history H is **conflict serialisable** if $C(H) \equiv_{CE}$ a serial history.

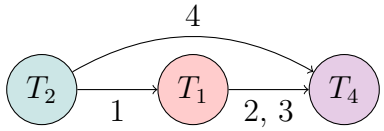
We can draw a **serialisation graph** $SG(H)$ which contains a node for each transaction in H, and an edge $T_i \to T_j$ if there exist some object o, for which a conflict $rw_i[o] \to rw_j[o]$ exists in H. If this graph is acyclic, then H is conflict serialisable. For example, consider the following history;

$$H_{cp} = r_1[b_{56}], r_2[b_{34}], w_2[b_{34}], w_1[b_{56}], r_4[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_4[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2, r_4[b_{67}], c_4$$

From this, we obtain the following conflicts;

1. $w_2[b_{34}] \to r_1[b_{34}]$
2. $w_1[b_{56}] \to r_4[b_{56}]$
3. $w_1[b_{34}] \to r_4[b_{34}]$
4. $w_2[b_{67}] \to r_4[b_{67}]$

Using this, we can draw the following acyclic graph, $S(H_{cp})$, hence we can say that H_{cp} is conflict serialisable, with an order T_2, T_1, T_4;



Recoverability

Serialisability is necessary for isolation and consistency of committed transaction, whereas recoverability is necessary for isolation and consistency when there are also aborted transactions. Some categories of histories are as follows;

- **recoverable execution (RC)**

  a history H has no transaction committing before another transaction from which it read
- **execution avoiding cascading aborts (ACA)**

  does not read from a non-committed transaction (does not allow any dirty reads to occur)
- **strict execution (ST)**

  does not read from a non-committed transaction, nor write over a non-committed transaction (no dirty reads nor writes)

There's some examples in the lecture slides, but they're quite tedious to copy out.

Topic 9 - Concurrency Control

The main approach we cover in this course is **two-phase locking (2PL)**, which directly implements the ideas of conflict previously mentioned. This approach uses locks to prevent problematic patterns.

Other approaches include time-stamping / version control, which only reads or writes objects with earlier timestamps, and aborts any changes when the object has a newer timestamp. Another approach is optimistic concurrency control which does nothing until the commit, at which point the history is inspected for problems.

2PL Protocol

The protocol is as follows;

1. read locks $rl[o], \ldots, r[o], \ldots, ru[o]$
2. write locks $wl[o], \ldots, w[o], \ldots, wu[o]$
3. has two phases

   i. growing phase
   ii. shrinking phase

   Note that more locks cannot be acquired during the shrinking phase.
4. refuse $rl_i[o]$ if $wl_j[o]$ is already held, and refuse $wl_i[o]$ if $rl_j[o]$ **or** $wl_j[o]$ are already held
5. if $rl_i[o]$ or $wl_i[o]$ is refused, then delay T_i

2PL works because we can re-time the history to have all operations occurring during the maximum lock period. This is CSR since all conflicts must be prevented during the maximum lock period.

Revisiting the phantom read anomaly, naive 2PL has an issue with inserting. Since no lock would be acquired for the new object (from the insert), the insert is able to happen without any refusal. The lock should've been done on the predicate, rather than the individual objects. There are some solutions;

- **table lock**

  The problem with a phantom read is due to the changing the data which matches a query, and here we acquire a read lock when performing a 'scan' of the table. This can produce needless conflicts, however it can be quite efficient of large parts of the table are being updated. As such, it would lock the entire table, for example $wl_7[a]$, rather than $wl_7[a_{101}]$ and $wl_7[a_{119}]$.

- **predicate locking / range locking**

  As the name implies, this locks on a predicate, which is more difficult to implement than the table lock. In this example, it would first acquire a read lock on the predicate, and then acquire individual write locks on each of the rows.

Scheduling

An **aggressive scheduler** delays taking locks as long as possible (only acquires the relevant lock before an operation). This is simple to implement, and maximises concurrency, however it may suffer delays later on. For example, if we needed to acquire a lock right at the end of our transaction (before all the other locks are released), but something else is holding it, any transaction waiting for the locks we're currently holding will also be delayed.

A **strict locking** variation prevents write locks from being released before the transaction ends (however read locks can be). This allows for deadlocks. Since write locks cannot be released, dirt reads / writes are avoided, and it is recoverable.
On the other hand a **strong strict locking** variation prevents any locks from being released before the transaction ends. This is simple to implement, and is also suitable for distributed transactions.

On the other hand, a **conservative scheduler** takes locks as soon as possible, and then releases them as it finishes. This removes the risk of delays later on, however the transaction may refuse to start (since it relies on all the locks being available). This prevents deadlock, but the problem is to figure out when to release locks. It is also not recoverable.

We can detect deadlocks with a WFG (waits-for graph). If a cycle is present in the graph, that means the database is in a deadlock state, and something must be aborted.

Anomalies Summarised

Using the same notation as before, we now have the following;

anomaly	pattern	prevented by
dirty write	$w_1[o] \prec w_2[o], w_2[o] \prec e_1$	strict 2PL
dirty read	$w_1[o] \prec r_2[o], r_2[o] \prec e_1$	strict 2PL
inconsistent analysis	$r_1[o_a] \prec w_2[o_a], w_2[o_b] \prec r_1[o_b]$	2PL
lost update	$r_1[o] \prec w_2[o], \prec w_2[o] \prec w_1[o]$	2PL
simple write skew	$r_1[o_a] \prec w_2[o_b], r_1[o_b] \prec w_2[o_a]$	2PL
write skew	$r_1[P_1] \prec w_2[x \in P_2], r_2[P_2] \prec w_1[y \in P_1]$	2PL (with pred. locks)
phantom read	$r_1[P_1] \prec w_2[x \in P_1], w_2[y \in P_2] \prec r_1[P_2]$	2PL (with pred. locks)

Isolation Levels

Note that we don't **always** need ACID properties. If we are doing something with an approximate result, such as obtaining an estimate, or some sort of overview, we can execute these transactions at a lower level of concurrency control.

- `READ UNCOMMITTED`

  This is the weakest level, which only prevents dirty writes.

- `READ COMMITTED`

  Prevents dirty reads and writes, which allows transactions to only read committed data. This is recoverable, but can still suffer inconsistent analysis.

- `SNAPSHOT`

  Allows transactions to see a view of the data at the start of the transaction, and write all data at the end of the transaction. This prevents every anomaly, other than write skew.

- `REPEATABLE READ`

  This allows inserts to tables which are already read, and prevents everything other than phantom reads.

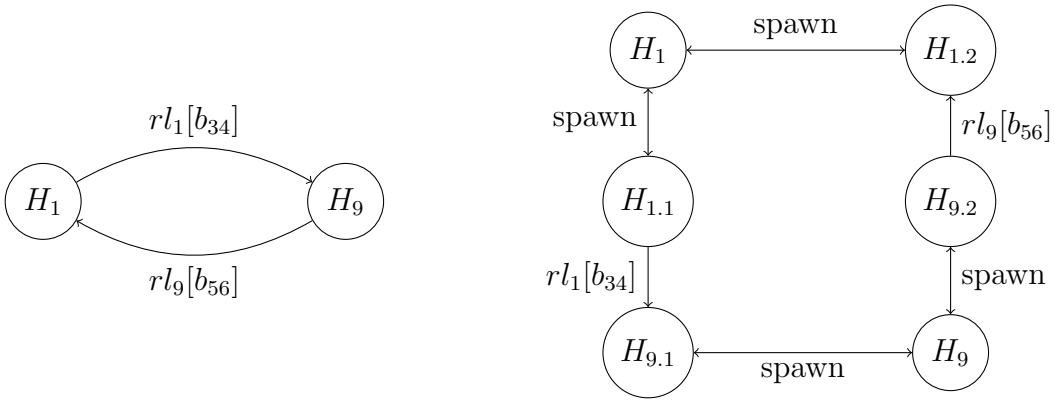
Topic 10 - Distributed Transactions

Distributed 2PL

With 2PL, we have the issue of synchronising the two phases across the different sites. Adopting a strong strict locking approach will amount to coordinating committment of transactions, which is the standard approach. If we're fragmenting data, we simply fragment the lock to the site where the relavent fragment of the data is present. Replication means that we need to, to some extent, copy the locks across the different sites.

WFG

We can convert a centralised WFG to sub-transactions by introducing bidirectional spawning edges;



When a local cycle is detected, the remote WFG is fetched and recombined to see if a cycle has been formed in the global WFG. If that the case, one of the transactions are aborted. Deadlock might have ocurred once a cycle has appeared at any node executing a distributed transaction.

Global 2PL and 2PC

The incorrect approach for performing 2PL globally would be to execute 2PL at each site. The correct approach would be to adopt strong strict locking at each site, and then the transactions can be ended with an global atomic commit.

In two-phase commit (2PC), we have the following (note that the coordinator could also operate from one of the servers, doesn't have to be a separate host);

service element	source	semantics
`C-PREPARE`	coordinator	get ready to commit
`C-READY`	server	ready to commit
`C-REFUSE`	server	not ready to commit
`C-COMMIT`	coordinator	commit the transaction
`C-ROLLBACK`	server	rollback the transaction
`C-RESTART`	either	try to return to start of transaction

If either of the servers sends a `C-REFUSE.request` (instead of `C-READY.request`) after a `C-PREPARE.request` from the server, a `C-ROLLBACK.request` is sent from the coordinator to both the servers, and a `C-ROLLBACK.response` is sent from the server **not** requesting the refusal. On the other hand, if both send `C-READY`, a `C-COMMIT` is sent from the coordinator.

If one of the servers crashes however, it can lead to blocking, since the non-crashed server will not know whether to commit or to abort.

Sending Locks

If we have fragmented data, the locks $wl_x[o]$ or $rl_x[o]$ must be sent to the host containing o. On the other hand, if we have replicated data, $w_x[o]$ must be sent to all hosts containing o. Similarly we can send send $r_w[o]$ to any host containing o.

As long as we send locks to the **majority** of sites, we will detect a write-write conflict. Therefore the number of locks we send;

$$j \geq \lceil \frac{n+1}{2} \rceil$$

On the other hand, to detect a read-write conflict, the number of locks we send, where k is the number of read locks, and j the number of write locks;

$$j \geq n - k + 1$$