Query Processing Models

Holger Pirk

Slides as of 10/02/22 09:24

Purpose of this lecture

What you know

- How data is stored
- How queries are logically represented

Today, you will

- learn How queries are represented physically (i.e., for the purpose of execution)
- learn how they are executed
- understand tradeoffs of processing models

Processing Models

What is a processing model?

The mechanism used to connect different operators

Why does it matter

- Different storage models optimizer for different bottlenecks:
  - Data Access
  - CPU
  - Query compilation time
  - ...

Recall:

Preliminaries

Tables

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

Storage Manager Interface

```
class StorageManager {
  map<string, Table> catalog;

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

Function Objects

Call them Lambdas, Function pointers, etc.

- They are pieces of code that are treated like data
  - They are basically pointers to an instruction
- You can assign them to variables...
- ...you can pass them as parameters to other functions...
- ...and you can evaluate/invoke/call them with arguments...
- ...and they will return a value

Python Syntax

```
aggregate = lambda v, t : v + t[0]
```

C++ Syntax

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

First thing to note: Volcano is a cool name!

Volcano Processing

- Volcano was a system that was very influential
  - Built in the 80s
  - Lots of focus on design practices – less on performance
- It had a lot of components:
  - The cascades query optimizer
  - A non-relational physical algebra
  - A query processing model
    - Let's talk about that

Volcano Design Goals

- Flexibility
- Clean Design
- Maintainability
- Developer Productivity
- A fair bit of tunability

Volcano Processing

Operators [Graefe: Query Evaluation Techniques for Large Databases]

Iterator	*Open*	*Next*	*Close*	Local State
Print	*open* input	call *next* on input; format the item on screen	*close* input	
Scan	open file	read next item	close file	open file descriptor
Select	*open* input	call *next* on input until an item qualifies	*close* input	
Hash join (without over-	allocate hash directory; *open* left	call *next* on probe input until a	*close* probe input; deallocate hash di-	hash directory

...

Volcano Processing

The operator interface

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

Implementation details

- Operators are connected using pointers
  - I will use the `unique_ptr` smart pointer

Scan

Purpose

Read a Table and return the contained tuples one by one

Implementation

```
struct Scan : Operator {
  Table input;
  size_t nextTupleIndex = 0;
  Scan(Table input) : input(input){};
  void open(){};
  optional<Tuple> next() {
    return nextTupleIndex < input.size() //
               ? input[nextTupleIndex++]
               : {};
  };
  void close(){};
};
```

- First observation: operators are stateful

Projection

Purpose

Transform a tuple into another tuple using a projection function

Implementation

```
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

Purpose

Return all tuples that satisfy a boolean predicate

Implementation

```
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() {
    for(auto dandidate = child->next(); dandidate.has_value; dandidate = child->next())
      if(predicate(dandidate))
        return dandidate;
    return {};
  };
  void close() { child->close(); };
};
```

Let's do one that has two inputs

Union

Purpose

Return all tuples from one child followed by all tuples from the other

Implementation

```
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();
  };
};
```

Let's do one that has two inputs and is challenging

Difference

Purpose

Read and buffer all tuples from the right side. After that, read all tuples from the left and return those that are not in the buffered right side.

Difference

Implementation

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

  void open() { // Read all tuples from one side into a buffer
    leftChild->open();
    rightChild->open();
    for(auto rightTuple = rightChild->next(); rightTuple.has_value; //
        rightTuple = rightChild->next())
      bufferedRight.push_back(rightTuple);
  };
  optional<Tuple> next() {
    for(auto nextCandidate = leftChild->next(); nextCandidate.has_value; //
        nextCandidate = leftChild->next()) {
      if(find(bufferedRight.begin(), bufferedRight.end(), nextCandidate) //
         == bufferedRight.end())
        return nextCandidate;
    }
    return {};
  };
  void close();
};
```

Difference is interesting...

. . . it forces me to read all inputs from one side before working on the other

We have a name for that:

a pipeline breaker

Pipeline Breakers

Definition

- A pipeline breaker is an operator that produces the first (correct) output tuple only after **all** input tuples from one of the sides have been processed

Cross Product

Purpose

Combine every tuple on the left with every tuple on the right.

Cross Product Implementation

First implementation...

```
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();
    for(auto rightTuple = rightChild->next(); rightTuple.has_value; //
        rightTuple = rightChild->next())
      bufferedRightTuples.push_back(rightTuple);
  };
  optional<Tuple> next(); // implementation on the next slide
  void close() {
    leftChild->close();
    rightChild->close();
  };
};
```

Cross Product Implementation

. . . first implementation

```
Tuple Cross::next() {
  if(currentBufferedRightOffset == bufferedRightTuples.size()) {
    currentBufferedRightOffset = 0;
    currentLeftTuple = leftChild->next();
  }
  if(!currentLeftTuple.has_value())
    return {};
  auto currentRightTuple = bufferedRightTuples[currentBufferedRightOffset++];
  return currentLeftTuple.concat(currentRightTuple);
}
```

This is clearly breaking the pipeline!

Can we do that without breaking the pipeline?

Pipelined Volcano Cross Product

Implementation...

```
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();
  };
  tuple next();
  void close() {
    leftChild->close();
    rightChild->close();
  };
};
```

Pipelined Volcano Cross Product

. . . implementation

```
tuple Cross::next() {
  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++]);
};
```

This is not breaking the pipeline!

Volcano Cross Product

Bottom line

- Whether operators are pipeline breakers depends on how they are implemented
- Some do not have pipelineable implementations
- Let's look at one of those...

Gouped Aggregation

Purpose

Group all tuples that are equal (given a projection function) and calculate one or more per-group aggregates

Implementation

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

Grouped Aggregation

Implementation

```
struct GroupBy : Operator {
  unique_ptr<Operator> child;
  vector<optional<Tuple>> hashTable; // size is magically known
  Projection getGroupKeys; // to extract the attributes we are grouping by (can be inlined)
  vector<AggregationFunction> aggregateFunctions;

  void open();

  int outputCursor = 0;
  optional<Tuple> next() {
    while(outputCursor < hashTable.size()) {
      auto slot = hashTable[outputCursor++];
      if(slot.has_value())
        return slot.value();
    }
    return {};
  };
  void close() { child->close(); }
};
```

Grouped Aggregation

Implementation (assuming single attribute groups)

```
size_t nextSlot(size_t value);
size_t hashTuple(Tuple t);
void GroupBy::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);

    if(!hashTable[slot].has_value) { // create new entry
      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();
  }
}
```

Using Volcano

Implementation

```
void moreInterestingQuery() {
  Table input{{1l, "Holger", "180 Queens Gate"},
              {2l, "Sam", "32 Vassar Street"},
              {3l, "Peter", "180 Queens Gate"}};
  auto plan =                                                          //
      make_unique<GroupBy>(                                            //
          make_unique<Select>(                                         //
              make_unique<Scan>(input),                                //
              [](auto t) { return t[2] == string("180 Queens Gate"); }), //
          [](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;
}
```

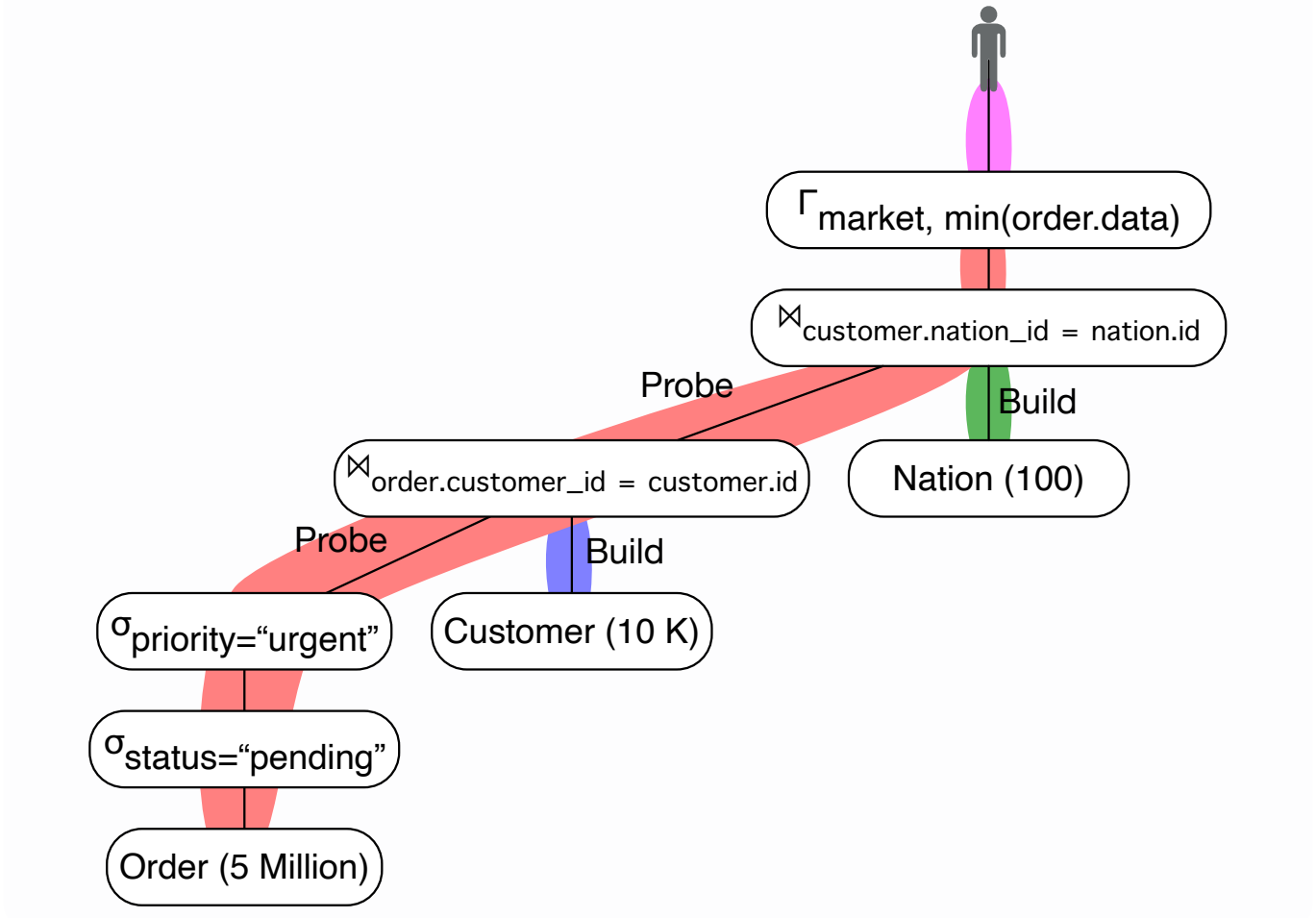
Advantages of Volcano

Elegant and simple implementation

- Our implementation is roughly 200 lines
- Good object-oriented design (Robert Chatley would be proud)
- Extensible
  - Adding new operators is easy (just adhere to the interface)
  - Takes advantage of the underlying language
- Good I/O Behavior:
  - Tuples are consumed as soon as they are produced

Pipelining

A plan with highlighted pipeline fragments



Estimating buffer I/O operations in Volcano...

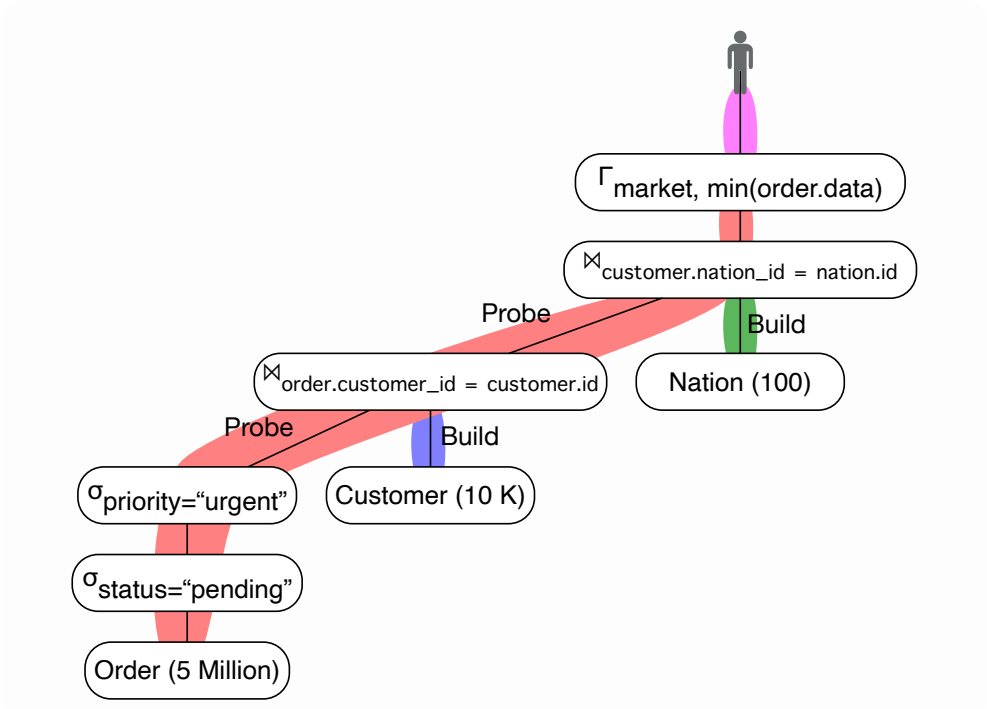
Scans read all pages of the relation

- number of pages calculated as we did in the Storage session

Pipeline Breakers: Writing intermediate buffers (`open` phase)

- If buffer and all other buffers in the fragment fit in memory: No I/O
- Otherwise:
  - buffer accessed sequentially: number of occupied pages (per pass)
  - buffer accessed randomly/out-of-order: one page access per tuple

. . . Estimating buffer I/O operations in Volcano. . .



. . . Estimating buffer I/O operations in Volcano. . .

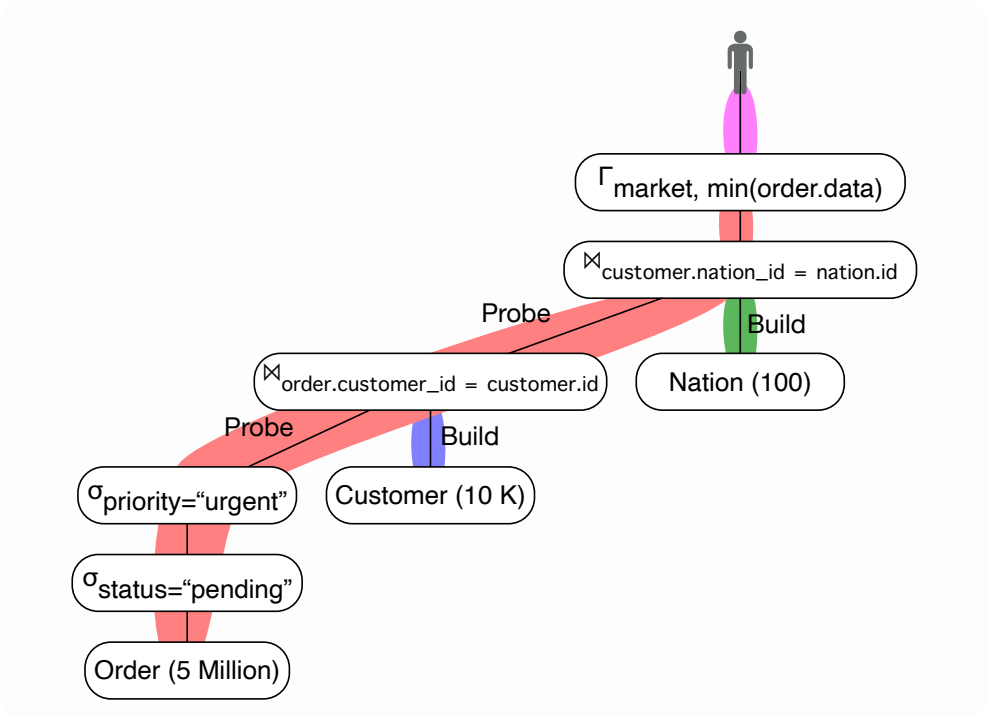
Pipeline Breakers: Reading intermediate buffers (`next` phase)

- If all buffers in the fragment (**combined**) fit in memory: No I/O
- if not, hash-probes: one page access per accessed tuple
- Otherwise: number of occupied pages (per pass)

Others

- No I/O

. . . Estimating buffer I/O operations in Volcano



How do we know if the buffers fit in memory?

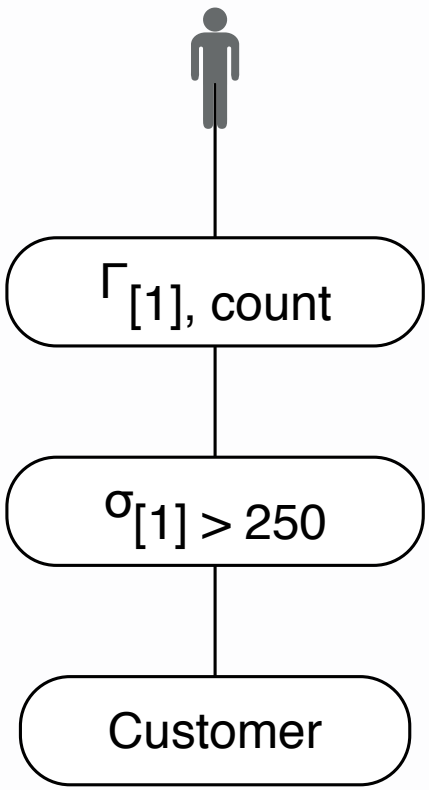
Rules

- Buffers need to hold data according to their algorithm & input
  - Nested loop buffers and sorted relations: exactly their input (number of tuples * tuple size)
    - Assume spanned pages
  - Hashtables are overallocated by a factor (assume two if not known)
- We assume perfect knowledge about the input and output cardinalities (a.k.a., an oracle)
  - Input buffer size: input cardinality times tuple size
  - Output buffer size: output cardinality times tuple size
- We know the memory/buffer pool size (may be given in bytes or in the number of pages)

Let's estimate some stuff!

Example: estimating buffer I/O operations in Volcano

Query



Customers (assuming spanned pages)

- 10.000 Tuples
- attributes: `id`, `name`, `address`, `nation`, `phone`, `accountNumber`
- `id`, `nation`, `phone`, `accountNumber` are int32
- `name`, `address` are dictionary-compressed (int32 keys)

Query Parameters

- Selection Selectivity: 30%, Grouping cardinality: 9
- Hashtable Overallocation Factor: 2
- Buffer Pool: 512KB, Page Size: 64 Byte

Example calculation

```
CustomerTableSize = 6 (*Attributes*) * 4 (*Bytes*) * 10000 (* tuples *);
CustomerTablePages = Ceiling[CustomerTableSize / 64] (*Bytes*); (* equation for spanned
↪  pages *)
CustomerScanCosts = CustomerTablePages;

GroupingCardinality = 9; (* Given *)
NumberOfAttributesInGroupingTable = 2; (* From plan *)

GroupingHashTableSize = Ceiling[2 * GroupingCardinality * NumberOfAttributesInGroupingTable
↪  * 4 (* Bytes *)]; (* = 144 Bytes*)
GroupingHashTableSize < BufferPoolSize; (* can be ignored *)

TotalPageIO == CustomerScanCosts
```

TotalPageIO == 3750

CPU Efficiency

What is the cost of a (sequential) memory access

- Back of the envelope calculation
  - My MacBook has 37.5 GB/s memory bandwidth and 4 cores @ 2.9 GHz
  - 9.375 GB/s per core
  - 3.23 Bytes per cycle (let's say about one integer)
- We better make sure we can process one integer per cycle

Function pointers

How they are evaluated by a CPU (roughly)

- The CPU stores the current instruction pointer (the `call`)
- The arguments are put on the execution stack
- The CPU instruction pointer is set to the address of the first instruction of the function code
    - This is called a **Jump (JMP)**
- The function is executed until it returns (there is a special return instruction)
- The instruction pointer is set to the instruction after the `call`

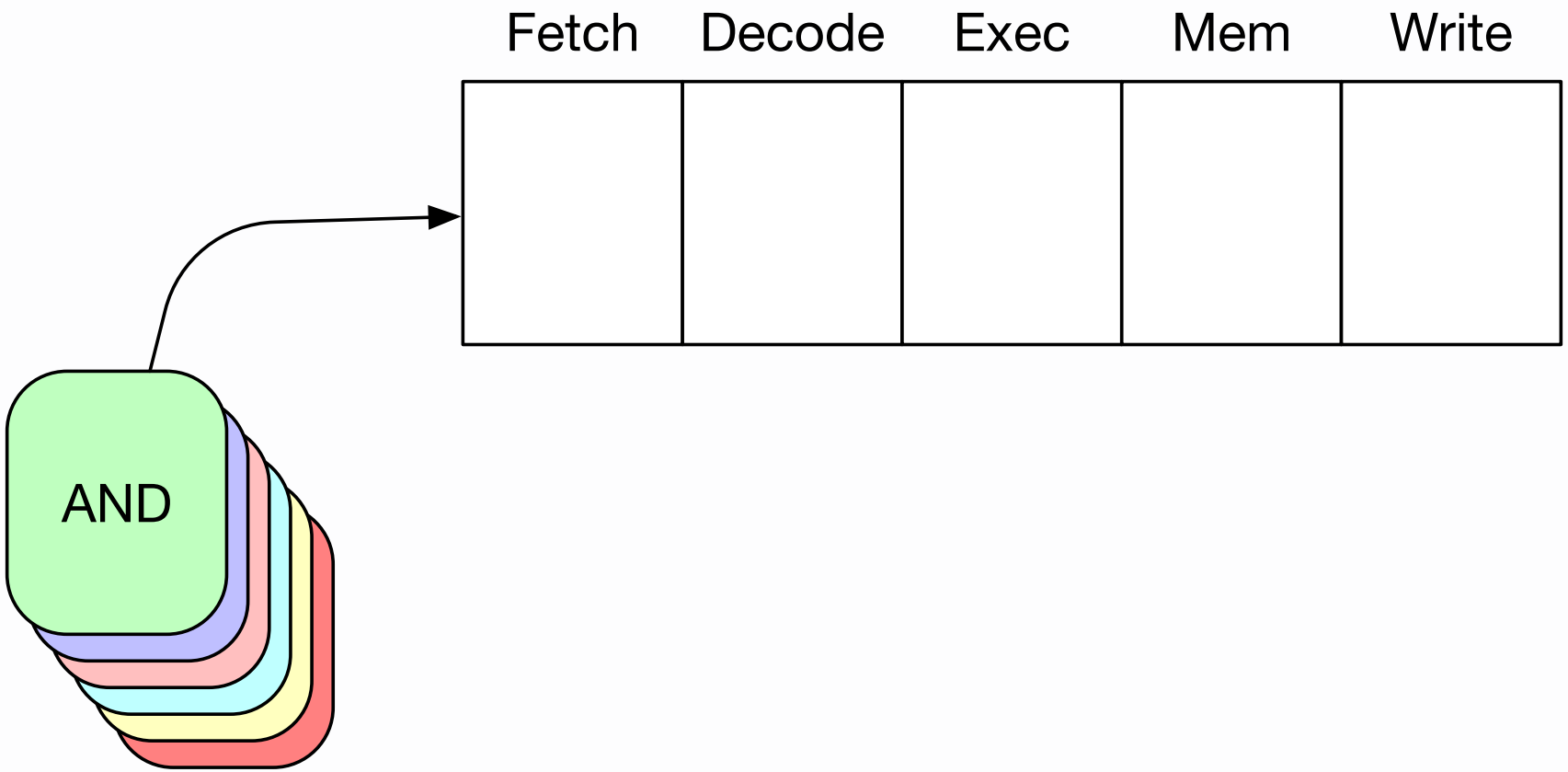
CPU execution pipelining (simplified)

Modern CPUs...

- ...execute instructions in stages...
- ...like `fetch`, `decode`, `execute`, `memory read`, `write result`.
- Instructions spend at least one cycle in each stage (for simplicity, let's say it is exactly one)...
- ...and move on to the next stage after every cycle.

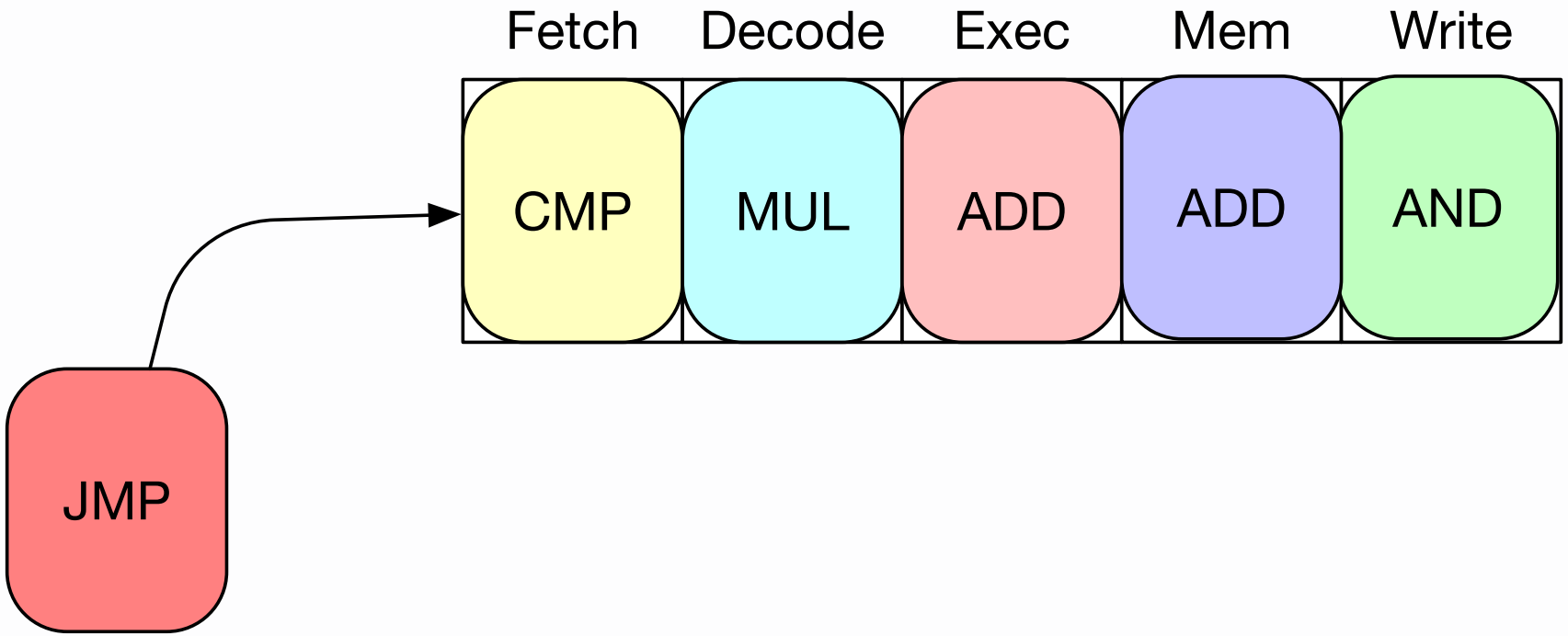
CPU execution pipelining (simplified)

An Empty CPU Pipeline



CPU execution pipelining (simplified)

A Filled CPU Pipeline

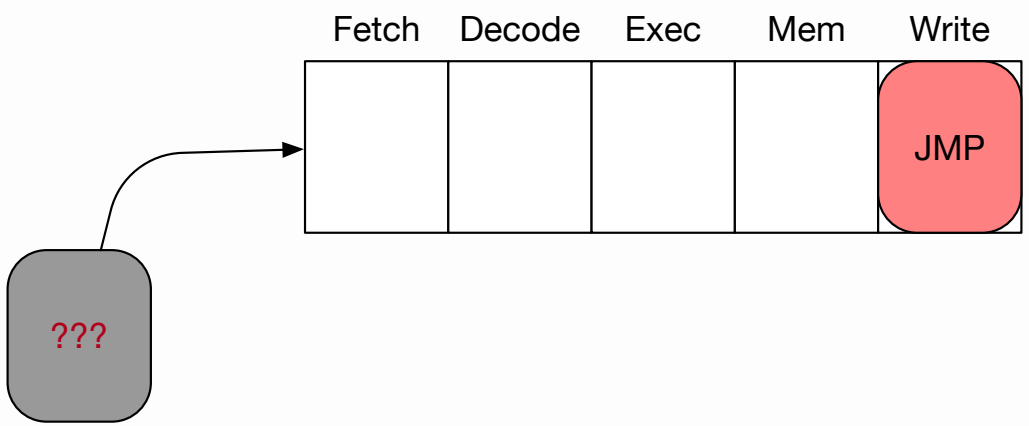


Function Pointers cause Pipeline Bubbles

Pipeline Bubbles (the technical term is Control Hazard)

- Remember: a Jump sets the instruction pointer to an arbitrary address
- The CPU needs to read the next instruction from this address
- Ergo: the next instruction can only be read once the jump is complete

A Control Hazard



The cost of a pipeline bubble

Impact

- Dependent on the length of the pipeline
  - My Macbook's CPU has around 15 stages
    - **That is 15 freaking cycles**

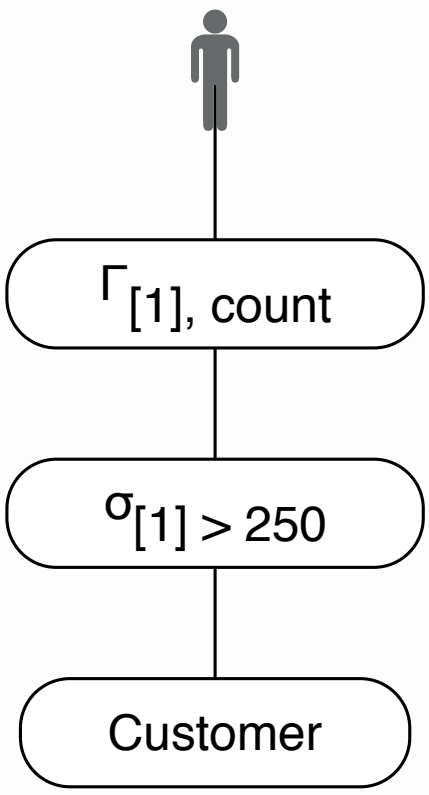
How many Function calls are there?

Per-tuple (we are counting inputs as it is easier)...

Scan	None, tuples are read straight from buffer
Selections/Projections	One to read the input, one to apply the predicate
Cross Product Inner	One to read the input
Cross Product Outer	One to read the input
Join (either side)	One to read the input (we can inline hash function for HJ as well as comparison for SMJ)
Group-By	One to read the input, one to calculate each new aggregate value
Output	One to extract it for output

Example: estimating function calls in Volcano

Query



Customers (assuming spanned pages)

- 10.000 Tuples
- attributes: `id`, `name`, `address`, `nation`, `phone`, `accountNumber`
- `id`, `nation`, `phone`, `accountNumber` are int32
- `name`, `address` are dictionary-compressed (int32 keys)

Query Parameters

- Selection Selectivity: 30%, Grouping cardinality: 9
- Hashtable Overallocation Factor: 2
- Buffer Pool: 512KB, Page Size: 64 Byte

Example: Calculation

```
FirstSelectSelectivity = .3;
CustomerTuples = 10000;
SelectFunctionCalls = CustomerTuples (*Tuples*) * 2 (*One to read input, one to apply
↪  selection*);
GroupingFunctionCalls = CustomerTuples * FirstSelectSelectivity * 2 (*One to read input, one
↪  to aggregate *);
FinalOutputExtraction = 9 (* One per group *);

FunctionCalls == ToString@DecimalForm@Total@{SelectFunctionCalls, GroupingFunctionCalls,
↪  FinalOutputExtraction}
```

```
FunctionCalls == 26009.
```

Now... With those numbers in hand

Volcano bottleneck analysis

Recall

- A function call costs 15 cycles
- We can read one integer per cycle

Calculation

Metric	Number	Cycles/Value	Total Cycles
Function Calls	26009	15	390135
Accessed 64-byte Pages	3750	16	60000

Volcano: The Bottom Line

What factor is bounding performance?

```
Updating from Wolfram Research server ...  Updating fro
```

Can we do something about that?

Yes!

Bulk Processing

The problem

- If CPU is the bottleneck...
- ...and function calls dominate CPU costs...
- ...can we process queries without any function calls
  - (or at least as few as possible)

The idea

- Turn *Control Dependencies* into *Data Dependencies*:
  - Instead of processing tuples right away, buffer them
  - Fill the buffer with lots of tuples
  - Pass the buffer to the next operator

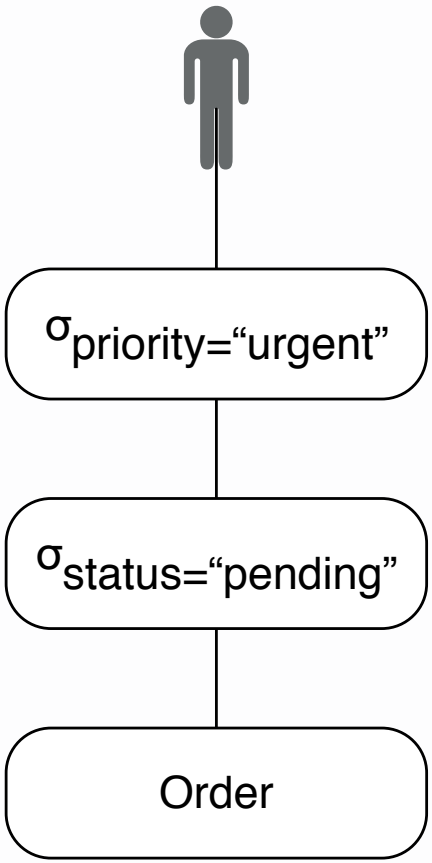
What Bulk Processing looks like

Operator

```
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();
};
```

What Bulk Processing looks like

Query Plan



Implementation

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

Bulk Processing

Bulk Processing means tight loops

- No function calls, no jumps
- Very CPU efficient
- **In Bulk Processing every operator is a pipeline breaker**
  - Similar rules apply that we used to calculate volcano I/O

Estimating buffer I/O operations in Bulk...

Rules

- Each operator reads all of its input sequentially
  - number of pages calculated as we did in the Storage session
- Each operator writes all of its output sequentially
  - number of pages calculated as we did in the Storage session

in addition...

. . . Estimating buffer I/O operations in Bulk

Reading/Writing temporary buffers (hashtables, etc.)

- If buffer fits in memory: No I/O
- Otherwise:
  - if buffer accessed sequentially: calculate number of occupied pages (per pass)
  - if buffer accessed randomly/out-of-order: one page access per accessed tuple

Note: All operators are cost-estimated completely independently!

Example: Selection

Parameters

- Selectivity: 25%
- 1M Tuples, 9 32-bit attributes each
- 512 KB Cache, 64 Byte Pages

Example: Selection

Calculation

```
Selectivity = .25;
CachelineSize = 64 (*Bytes*);
BufferPoolCapacityInPages = (512*1024)/64;
TableSizeInTuples = 1000000;
TableTupleSize = 9 (*Attributes*) * 4 (*Bytes*);

TableSize = TableTupleSize * TableSizeInTuples;
TablePages = Ceiling[TableSize / CachelineSize]; (* Spanned Pages *)

SelectionInputInPages = TablePages;
SelectionOutputInPages = Ceiling[TableSize * Selectivity / CachelineSize];
SelectionIO == SelectionInputInPages + SelectionOutputInPages
```

```
SelectionIO == 703125
```

By-Reference Bulk Processing

Saving Bandwidth

- We are copying a lot of data around
  - This is a classic computing problem with a classic solution:
  - Call by reference
- Instead of producing tuples, we produce their IDs (32-bit positions in their buffer)
- When processing a tuple, we always use the ID to look up the actual value
  - Lookup costs are the same as they are for a hashtable without conflicts

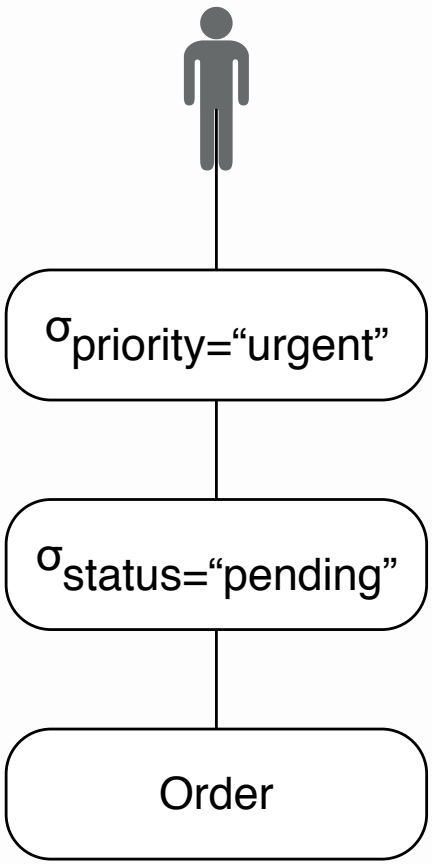
By-Reference Bulk Processing

Select operator implementation

```
int select(vector<int>& outputBuffer, optional<vector<int>> const& candidatePositions, //
           int predicate, int attributeOffset, vector<Tuple> const& underlyingRelation) {
  if(!candidatePositions.has_value()) { // first selection in the plan -> all tuples
  ↪  candidates
    for(size_t i = 0; i < underlyingRelation.size(); i++) {
      if(underlyingRelation[i][attributeOffset] == predicate)
        outputBuffer.push_back(i);
    }
  } else { // later selection in the plan -> some tuples potentially not candidates
    for(size_t i = 0; i < candidatePositions->size(); i++) {
      if(underlyingRelation[(*candidatePositions)[i]][attributeOffset] == predicate)
        outputBuffer[outputCursor++] = (*candidatePositions)[i];
    }
  }
  return outputCursor;
}
```

By-Reference Bulk Processing

Query Plan



Implementation

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

The rules for input and output have just become more tricky!

Estimating buffer I/O operations in by-reference bulk

Rules

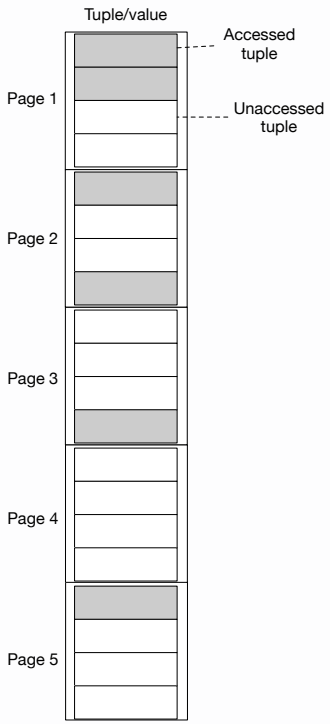
- Each operator reads all of its candidate buffer sequentially
  - number of pages calculated as we did in the Storage session
- Each operator writes all of its candidate buffer sequentially
  - number of pages calculated as we did in the Storage session

In addition

Each operator "resolves" the candidate references by looking up the values in the base relation

Calculating Page Access probability

Visualization



Explanation

- Selectivity s is the percentage of tuples being touched, n the number of tuples on a page
- Assume uniformly distributed values
- What is the probability of any one of them being touched?
  - $p(s, n) = 1 - (1 - s)^n$
- Number of pages:

```
ProbabilityOfAccessingPage[s_,n_] := 1-(1-s)^n;
PageFaults := ProbabilityOfAccessingPage[
        FirstSelectSelectivity, AverageBaseTableTuplesPerPage]
        * BaseTablePages
```

The rest still holds:

Estimating buffer I/O operations in by-reference bulk

Writing temporary buffers (hashtables, etc.)

- If buffer fits in memory: No I/O
- Otherwise:
  - if buffer accessed sequentially: calculate number of occupied pages (per pass)
  - if buffer accessed randomly/out-of-order: one page access per accessed tuple

Reading temporary buffers

- If buffer fits in memory: No I/O
- if not, hash-probes: one page access per accessed tuple
- Otherwise: sequential I/O over buffer

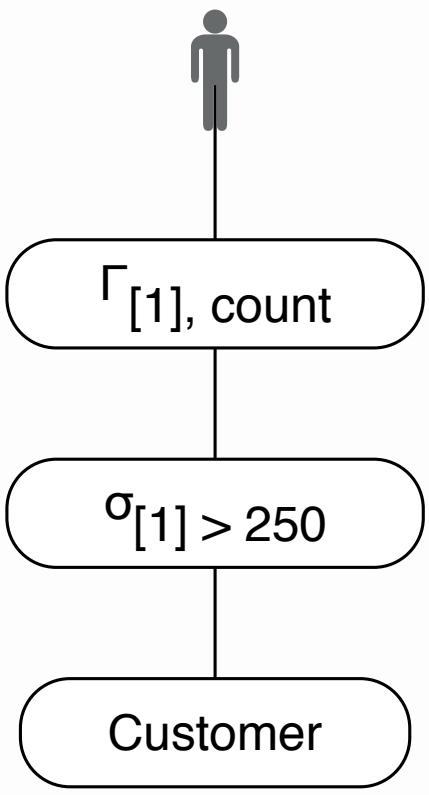
By-Reference Bulk Processing of Decomposed Data

Saving even more bandwidth

- Every operator processes exactly one column of a tuple
- In N-ary, storage, values of a tuple are co-located on a page
  - i.e., you always pay for all values on a page (even if you only process one)
  - these useless values also occupy space in the buffer pool/cache
- DSM fixes both of these problems
  - **DSM was introduced to databases as a consequence of Bulk Processing**
  - **Not the other way around**
  - I have it on pretty good authority!

Example: estimating function calls in Volcano

Query



Customers (assuming spanned pages)

- 10.000 Tuples
- attributes: `id`, `name`, `address`, `nation`, `phone`, `accountNumber`
- `id`, `nation`, `phone`, `accountNumber` are int32
- `name`, `address` are dictionary-compressed (int32 keys)

Query Parameters

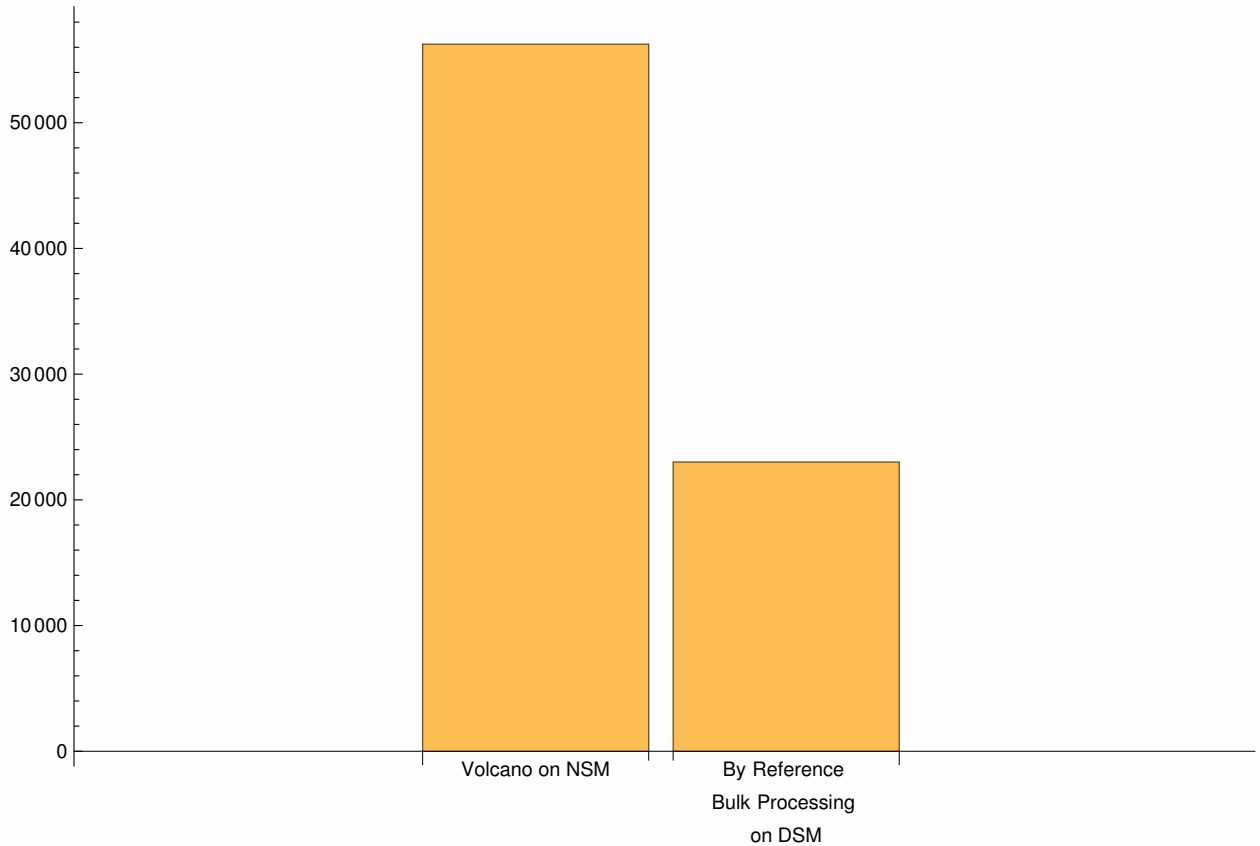
- Selection Selectivity: 30%, Grouping cardinality: 9
- Hashtable Overallocation Factor: 2
- Buffer Pool: 512KB, Page Size: 64 Byte

By-Reference Bulk Processing of DSM Data

```
ProbabilityOfAccessingPage[s_, n_] := 1 - (1 - s) ^ n;

SelectSelectivity = .3;
CachelineSize = 64(*Bytes*);
BufferPoolCapacityInPages = (512 * 1024) / 64;
OrderTableSizeInTuples = 10000;
OrderTupleAttributeSize = 1(*Attributes*)*4(*Bytes*);

OrderTableColumnSize = OrderTupleAttributeSize * OrderTableSizeInTuples;
OrderTableColumnPages = Ceiling[OrderTableColumnSize / CachelineSize];
(*Spanned Pages*)

SelectionInputInPages = OrderTableColumnPages;
SelectionOutputInTuples = OrderTableSizeInTuples * SelectSelectivity;
SelectionOutputInPages = Ceiling[SelectionOutputInTuples * 4(*Bytes*) / CachelineSize];
SelectionIO = SelectionInputInPages; (* SelectionOutputInPages < BufferPoolCapacityInPages
↪  *)

GroupingHashTableSize =
    Ceiling[2 * GroupingCardinality * NumberOfAttributesInGroupingTable * 4(*Bytes*)];
(*SelectionOutputInPages + GroupingHashTableSize < BufferPoolCapacityInPages*)

OrderTuplesPerPage = CachelineSize / OrderTupleAttributeSize;
GroupingIO = Ceiling[ProbabilityOfAccessingPage[SelectSelectivity, OrderTuplesPerPage] *
                     OrderTableColumnPages](*Input 2 *) +
             Ceiling[9 * 2(*Attributes*)*4(*Bytes*) / CachelineSize](*Output*);

TotalPageIO == ToString @DecimalForm[SelectionIO + GroupingIO] == 1438
```

The bottom line on bulk processing

Execution cost by processing model (for our query)



Now the big question:

Why am I telling you now?

Is it too late to do something like that for the competition?

Turns out: you do not have that problem...

. . . because you know the query at compile-time

Further reading

Bulk-processing lead to "Vectorization"

Zuckowski. "Balancing vectorized query execution with bandwidth-optimized storage." PhD Thesis, 2009 Sompolski et al. "Vectorization vs. compilation in query execution." ACM DaMoN, 2011

Thank you

Provide feedback, please!

https://co572.pages.doc.ic.ac.uk/feedback/processingmodels

Get the slides online

https://co572.pages.doc.ic.ac.uk/decks/ProcessingModels.pdf