Set-Oriented Query Processing
Motivation

During query processing, the DBMS tries to process whole sets of data items at a time

- “manual” programming is usually record oriented
- e.g., compare two records
- easy to understand, but this does not scale

Consider: intersecting two lists

- breaking it down into record-level operators is inefficient
- compares each record with each other record
- $O(n^2)$
- considering the complete lists in one step is more efficient
- $O(n \log n)$
Motivation (2)

Set-oriented processing has several advantages

- data can be pre-processed before processing
- sorting/hashing/index structures etc.
- amortizes over the set
- leads to more efficient algorithms
- easier to cope with memory limitations etc.
- easier parallelism
- ...

Algorithms tend to become more scalable, but also more involved.
The Algebraic Model

Query processing is usually expressed by relational algebra

- operators consumes zero or more relations, and produce one output relation
- inherently set (or rather: bag) oriented
Implementing the Algebraic Model

Operators are specified in a query agnostic manner:

- intersect
  - left
  - right
  - compare

Operator does not understand the query semantic. It only knows:

- *left* will produce a result set
- *right* will produce a result set
- *compare* compares two elements

Note: a scalable implementation will need more (e.g., hashLeft, hashRight), we ignore this for now.
Implementing the Algebraic Model (2)

The algebraic operators define the **abstract logic** of query processing primitives. The query specific parts are hidden in **subscripts**.

In particular:

- operators do not “know” the data types or byte size of input tuples
- they do not “understand” the content of a tuple
- they only specify the data flow and the control flow
- all query dependent operations are delegated to helper subscripts
- keeps the operator itself very generic

Note: sometimes operators are hinted with query specific info (e.g., a fixed tuple size) for performance reasons, but this is only a minor variation.
Implementing the Algebraic Model (3)

Example: intersectSorted(left,right,compare)

\[ t_1 = \text{next tuple from } left \]
\[ n = right \]

while input is not exhausted
  \[ \text{if } n = left \]
  \[ t_1 = \text{next tuple from } left \] else
  \[ t_2 = \text{next tuple from } right \]
  \[ c = \text{compare} (t_1, t_2) \]
  \[ \text{if } c = 0 \]
  \[ \text{store } t_1 \text{ as result} \]
  else if \[ c < 0 \]
  \[ n = left \]
  else
  \[ n = right \]

The code is independent from the concrete query.
Operator Composition

- each operator produces a set (bag/stream) of result tuples
- operators consume zero or more input sets
- usually assume nothing about their input
- therefore can be combined in an arbitrary manner
- very flexible
Operator Interface
Option 1: Full Materialization

Every operator materializes its output. The input is always read from a materialized state.

Advantages:
- easy to implement
- can handle surprises concerning intermediate result sizes (dynamic plans)
- advanced techniques like parallelization, result sharing, etc. are simple

Disadvantages:
- materialization is expensive
- in particular if data is larger than main memory

Few systems use this approach, but some do (MonetDB).
Operator Interface (2)

Option 2: Iterator Model

Each operator produces a tuple stream on demand. The input is iterated over.

Advantages:
- data is pipelined between operators
- avoids unnecessary materialization
- flexible control flow
- easy to implement

Disadvantages:
- millions of virtual function calls
- poor locality

The standard model. Widely used.
The iterator model usually offers the following interface:

- open
- next
- close

Repeated calls to next produce the output stream.

Internally, operators maintain a complex state to offer the iterator interface.
Operator Interface (4)

How to pass data from one operator to the other?
- the data itself is opaque
- as a consequence, it cannot be passed (easily) by value

Alternative 1: pass tuple pointers
- the real data resides on a page/in the buffer
- operators are only passed pointers to the data

Alternative 2: not at all
- there is a global data space (“registers”)
- subscript functions operate on these registers
- the operators never touch the data directly

Alternative 2 is more generic, and can cope better with computed columns.
Operator Interface (5)

Option 3: blockwise processing
Each operator produces a tuple stream, but not tuple-by-tuple but as a stream of larger chunks.

Advantages:
- far fewer function calls
- better code and data locality

Disadvantages:
- additional materialization overhead
- consumes memory bandwidth
- control flow not as flexible
Operator Interface (6)

Option 4: pushing tuples up
Each operator pushes produced tuples towards the consuming operators.

Advantages:
- operator logic is concentrated in a few loops
- good code and data locality
- pipelining etc. still possible
- support for DAG-structured plans

Disadvantages:
- some restrictions in control flow
- code generation more involved
Examples - Full Materialization

scan($R$)
   // no-op, all operators read their input
   return $R$

select($R,p$)
   $R' =$ new temporary relation
   for each $t \in R$
      if $p(t)$
         append $t$ to $R'$
   return $R'$

cross($R_1,R_2$)
   $R' =$ new temporary relation
   for each $t_1 \in R_1$
      for each $t_2 \in R_2$
         append $t_1 \circ t_2$ to $R'$
   return $R'$
Examples - Iterator Model

```cpp
class Scan
    in, tid, limit

Scan::open(R)
    in = R
    tid = 0
    limit = |R|

Scan::next()
    if tid >= limit
        return false
    load tuple t from in at position tid
    tid = tid + 1
    return true
```
class Select
  in, p

Select::open(in, p)
  this.in = in
  this.p = p

Select::next(in, p)
  while in.next()
    if p()
      return true
  return false
Examples - Iterator Model (3)

```cpp
class Cross
    left, right, step
Cross::open(left, right)
    this.left = left
    this.right = right
    step = true
Cross.next()
while true
    if step
        if not left.next()
            return false
        right.open()
        step = false
    if right.next()
        return true
    step = true
```
Examples - Blockwise Processing

class Scan
    in, tid, limit

Scan::open(R)
    in = R
    tid = 0
    limit = |R|

Scan::next()
    C = \min(limit - tid, 1000)
    R' = tuple array of size C
    for i = 0...C - 1
        load tuple R'[i] from in at position tid + i
    tid = tid + C

return R'
Examples - Blockwise Processing (2)

class Select
    in, p
Select::open(in, p)
    this.in = in, this.p = p
Select::next(in, p)
    while true
        R' = in.next()
        if |R'| = 0
            return R'
        w = 0
        for i = 0...|R'| - 1
            R'[w] = R'[i]
            w = w + p(R'[w])
        R'.length = w
        if |R'| > 0
            return R'
Examples - Blockwise Processing (3)

```java
class Cross
    left, right, c_L, l_L, R_L, c_R, l_R, R_R

Cross::open(left, right)
    this.left = left
    this.right = right
    step = true
    c_L = l_L = c_R = r_R = 0

Cross.next()
    R' = tuple array of size 1000, w = 0
```
Examples - Blockwise Processing (4)

while true
    while true
        while $c_R = l_R$
            if $c_L \geq l_L$
                $R_R = \text{right}.\text{next}()$
                if $|R_R| = 0$
                    $R_L = \text{left}.\text{next}()$
                    if $|R_L| = 0$
                        $R'.\text{length} = w$, return $R'$
                        right.\text{rewind}(), $R_R = \text{right}.\text{next}()$
                        $c_L = 0$, $l_L = |R_L|$
            else $c_L = c_L + 1$
        $c_R = 0$, $l_R = |R_R|$
        $R'[w] = R_L[c_L] \circ R_R[c_R]$
        $c_R = c_R + 1$, $w = w + 1$
        if $w = |R'|$
            return $R'$
Examples - Push

class Scan
   consumer, R

Scan::open(consumer, R)
   this.consumer = consumer
   this.R = R

Scan::produce()
   for each t in R
      consumer.consume(t)
Examples - Push (2)

```cpp
class Select
    in, consumer, p

Select::open(in, consumer, p)
    this.in = in, this.consumer = consumer, this.p = p

Select::produce()
    in.produce()

Select::consume(t)
    if p(t)
        consumer.consume(p)
```
Examples - Push (3)

```cpp
class Cross
    left, right, consumer, t_L

Cross::open(left, right, consumer)
    this.left = left, this.right = right, this.consumer = consumer

Cross::produce()
    left.produce()

Cross::consumeFromLeft(t)
    t_L = t
    right.produce()

Cross::consumeFromRight(t)
    consumer.consume(t_L \circ t)
```
Additional Functionality

We ignored the close function so far
- releases allocated resources

Other functionality implemented or used by operators:
- rewind/rebind
- memory management
- spooling intermediate results
Implementing Subscripts

The operators are query independent, but the subscripts are not

- cover the query-specific parts of the query
- attribute access (e.g., \( x.a \))
- predicates (e.g., \( a=b \))
- computations (e.g., \( \text{sum(amount*(1+tax))} \))
- ...

Must be implemented, too

- different for every query
- but usually relatively simple
- complexity much lower than for operators
Implementing Subscripts (2)

Option 1: interpreter objects

Subscripts are assembled from interpreter objects.

- very flexible
- easy to implement
- widely used
- but: many virtual function calls

Val AccessInt::eval(char* ptr)

```cpp
return *((int*)(ptr+ofs));
```

Val CompareEqInt::eval(char* ptr)

```cpp
return left->eval(ptr).intValue==right->eval(ptr).intValue
```
Implementing Subscripts (3)

Option 2: virtual machines

Subscripts are compiled into instructions for a virtual machine.

- more efficient than interpreter objects
- but also more complex
- requires a compiler to byte code

```c
while (true) switch ((++op)->cmd) {
    case Cmd::AccessInt:
        reg[op->out] = *((*int)(ptr+op->val));
        break;
    case Cmd::CompareEqInt:
        break;
    ...
}
```
Implementing Subscripts (4)

Option 3: pre-compiled fragments

Subscripts are expressed as combination of pre-compiled fragments.
- each fragment performs a number of operations
- quite efficient (vectorization)
- but usually only applicable for column stores

```c
CompareEqInt(unsigned len, int* col1, int* col2, bool* result)
    for (unsigned index=0; index!=len; ++index)
        result[index]=col1[index]==col2[index]
```
Implementing Subscripts (5)

Option 4: generated machine code

Subscripts are at runtime compiled into native machine code.

- the most efficient alternative
- but also the most difficulty
- portability is an issue
- we will look at this in the Section Code Generation

... 

movq 72(%rsp), %rax
movl (%rax,%r12,4), %r13d
movq 120(%rsp), %rax
movl (%rax,%r12,4), %edi
cmpl %r13d,%edi

...
Pipelining

As mentioned, most approaches try to avoid copying data between operators

- this is called *pipelining*
- operators that do materialize their input are called *pipeline breakers*
- operators are consume their input completely before processing are called *full pipeline breakers*
- some binary operators are pipeline breakers on only one side

This behavior has implications regarding other operators.
Pipelining (2)

Some effects of different pipeline behavior

- if a pipeline break is between source and sink the original data is no longer accessible
  - relevant for lazy attribute access/TID join/string representations etc.
  - the system must plan defensively
- if a full pipeline breaker is between two operators both are decoupled
  - the full pipeline break breaks the plan into fragments
  - can be executed independent from each other
  - relevant for scheduling
- ...

The code generation must know the pipeline behavior of operators.
Parallelization

How can we exploit multiple cores during query processing?

- inter-query parallelism is simple
- intra-query parallelism is much harder
- independent parts of the query can be executed in parallel (see: full pipeline breaker)
- parallelizing individual operators is more difficult
- usual strategy: partition the input

We will discuss this later in more detail.