Code Generation

Motivation

For good performance, the operator subscripts have to be compiled

- either byte code
- or machine code
- generating machine code is more difficult but also more efficient

Machine code has portability problems

- code generation frameworks hide these
- some well known kits: LLVM, libjit, GNU lightning, ...
- greatly simplify code generation, often offer optimizations

LLVM is one of the more mature choices.

LLVM

Distinct characteristics

- unbounded number of registers
- SSA form
- strongly typed values

```
define i32 @fak(i32 %x) {
      %1 = icmp ugt i32 %x, 1
      br i1 %1, label %L1, label %L2
L1:   %2 = sub i32 %x, 1
      %3 = call i32 @fak(i32 %2)
      %4 = mul i32 %x, %3
      br label %L3
L2:   br label %L3
L3:   %5 = phi i32 [ %4, %L1 ], [ 1, %L2 ]
      ret i32 %5
}
```

Compiling Scalar Expressions

- all scalar values are kept in LLVM registers
- additional register for NULL indicator if needed
- most scalar operations (=, +, -, etc.) compile to a few LLVM instructions
- C++ code can be called for complex operations (`like` etc.)
- goal: minimize branching, minimize function calls

The real challenge is integrating these into set-oriented processing.

Data-Centric Query Execution

Why does the iterator model (and its variants) use the operator structure for execution?

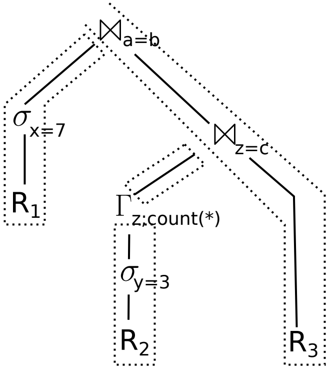
- it is convenient, and feels natural
- the operator structure is there anyway
- but otherwise the operators only describe the data flow
- in particular operator boundaries are somewhat arbitrary

What we really want is **data centric** query execution

- data should be read/written as rarely as possible
- data should be kept in CPU registers as much as possible
- the code should center around the data,
  not the data move according to the code
- increase locality, reduce branching

Data-Centric Query Execution (2)

Example plan with visible pipeline boundaries:



- data is always taken out of a pipeline breaker and materialized into the next
- operators in between are passed through
- the relevant chunks are the pipeline fragments
- instead of iterating, we can push up the pipeline

Data-Centric Query Execution (3)

Corresponding code fragments:

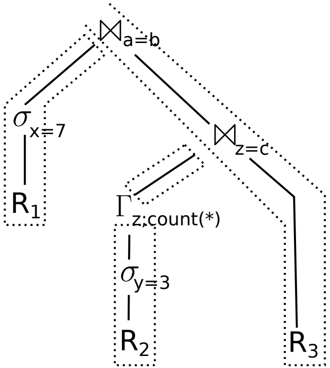
initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z

for each tuple t in R_1
  if $t.x = 7$
    materialize t in hash table of $\bowtie_{a=b}$

for each tuple t in R_2
  if $t.y = 3$
    aggregate t in hash table of Γ_z

for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$

for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}[t_3.c]$
    for each match t_1 in $\bowtie_{a=b}[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$



Data-Centric Query Execution (4)

Basic strategy:

1. the producing operator loops over all materialized tuples
2. the current tuple is loaded into CPU registers
3. all pipelining ancestor operators are applied
4. the tuple is materialized into the next pipeline breaker

- tries to maximize code and data locality
- a tight loops performs a number of operations
- memory access in minimized
- operator boundaries are blurred
- code centers on the data, not the operators

Producing the Code

Code generator mimics the produce/consume interface

- these methods do not really exist, they are conceptual constructs
- the *produce* logic generates the code to produce output tuples
- the *consume* logic generates the code to accept incoming tuples
- not clearly visible within the generated code

Producing the Code (2)

```
void HJTranslatorInner::produce(CodeGen& codegen,Context& context) const
{
  // Construct functions that will be be called from the C++ code
  {
    AddRequired addRequired(context,getCondiution().getUsed().limitTo(left));
    produceLeft=codegen.derivePlanFunction(left,context);
  }
  {
    AddRequired addRequired(context,getCondiution().getUsed().limitTo(right));
    produceRight=codegen.derivePlanFunction(right,context);
  }

  // Call the C++ code
  codegen.call(HashJoinInnerProxy::produce.getFunction(codegen),
    {context.getOperator(this)});
}
void HJTranslatorInner::consume(CodeGen& codegen,Context& context) const
{
```

Producing the Code (3)

```
// Left side
if (source==left) {
  // Collect registers from the left side
  vector<ResultValue> materializedValues;
  matHelperLeft.collectValues(codegen,context,materializedValues);

  // Compute size and hash value
  llvm::Value* size=matHelperLeft.computeSize(codegen,materializedValues);
  llvm::Value* hash=matHelperLeft.computeHash(codegen,materializedValues);

  // Materialize in hash table, spools to disk if needed
  llvm::Value* ptr=codegen.callBase(HashJoinProxy::storeLeftInputTuple,
    {opPtr,size,hash});
  matHelperLeft.materialize(codegen,ptr,materializedValues);
```

Producing the Code (4)

```
// Right side
} else {
  // Collect registers from the right side
  vector<ResultValue> materializedValues;
  matHelperRight.collectValues(codegen,context,materializedValues);

  // Compute size and hash value
  llvm::Value* size=matHelperRight.computeSize(codegen,materializedValues);
  llvm::Value* hash=matHelperRight.computeHash(codegen,materializedValues);

  // Materialize in memory, spools to disk if needed, implicitly joins
  llvm::Value* ptr=codegen.callBase(HashJoinProxy::storeRightInputTuple,
    {opPtr,size});
  matHelperRight.materialize(codegen,ptr,materializedValues);
  codegen.call(HashJoinInnerProxy::storeRightInputTupleDone,{opPtr,hash});
}
}
```

Producing the Code (5)

```
void HJTranslatorInner::join(CodeGen& codegen,Context& context) const
{
   llvm::Value* leftPtr=context.getLeftTuple(),*rightPtr=context.getLeftTuple();
   // Load into registers. Actual load may be delayed by optimizer
   vector<ResultValue> leftValues,rightValues;
   matHelperLeft.dematerialize(codegen,leftPtr,leftValues,context);
   matHelperRight.dematerialize(codegen,rightPtr,rightValues,context);

   // Check the join condition, return false for mismatches
   llvm::BasicBlock* returnFalseBB=constructReturnFalseBB(codegen);
   MaterializationHelper::testValues(codegen,leftValues,rightValues,
      joinPredicateIs,returnFalseBB);
   for (auto iter=residuals.begin(),limit=residuals.end();iter!=limit;++iter) {
      ResultValue v=codegen.deriveValue(**iter,context);
      CodeGen::If checkCondition(codegen,v,0,returnFalseBB);
   }
   // Found a match, propagate up
   getParent()->consume(codegen,context);
```

Parallel Query Execution

Parallelism

Why parallelism

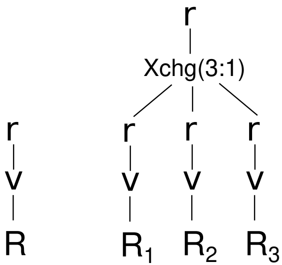
- multiple users at the same time
- modern server CPUs have dozens of CPU cores
- better utilize high-performance IO devices

Forms of parallelism

- inter-query parallelism: execute multiple queries concurrently
  - map each query to one process/thread
  - concurrency control mechanism isolates the queries
  - except for synchronization that parallelism is "for free"
- intra-query parallelism: parallelize a single query
  - horizontal (bushy) parallelism: execute independent sub plans in parallel (not very useful)
  - vertical parallelism: parallelize operators themselves

Vertical Parallelism: Exchange Operator

- optimizer statically determines at query compile-time how many threads should run
- instantiates one query operator plan for each thread
- connects these with "exchange" operators, which encapsulate parallelism, start threads, and buffer data
- relational operator can remain (largely) unchanged
- often (also) used in a distributed setting



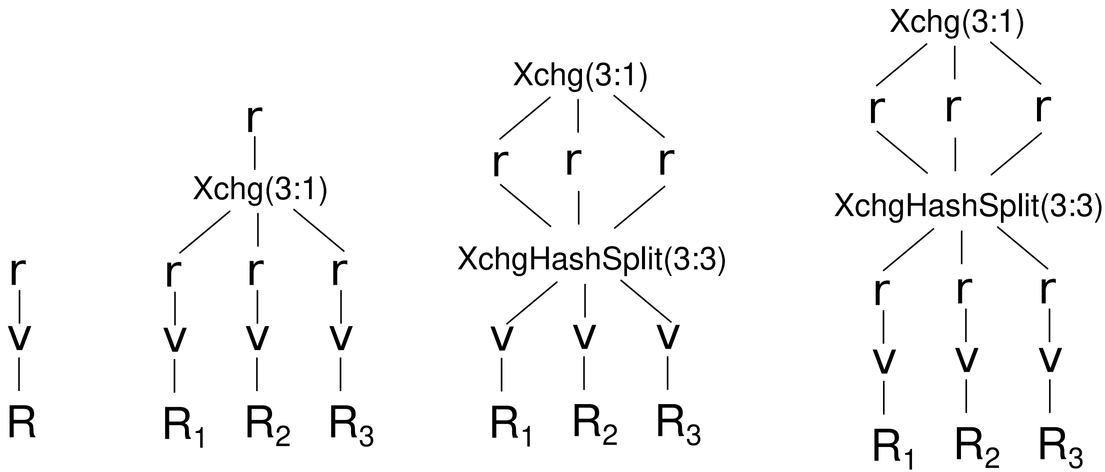
Exchange Operator Variants

- Xchg(N:M) N input pipelines, M output pipelines

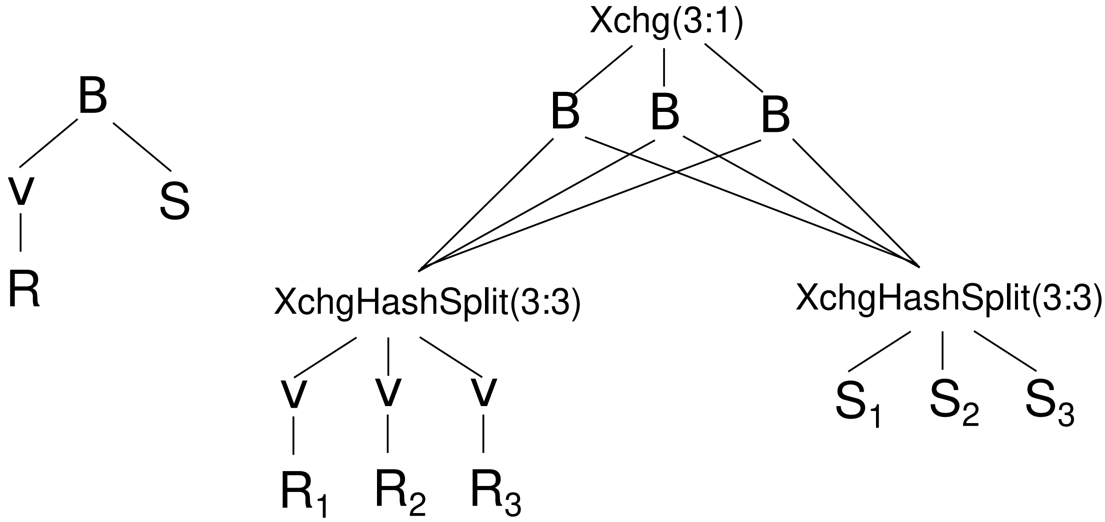
Many useful variants

- XchgUnion(N:1) specialization of Xchg
- XchgDynamicSplit(1:M) specialization of Xchg
- XchgHashSplit(N:M) split by hash values
- XchgBroadcast(N:M) send full input to all consumers
- XchgRangeSplit(N:M) partition by data ranges

Aggregation with Exchange Operators (3-way parallelism)



Join with Exchange Operators (3-way parallelism)



Disadvantages of Exchange Operators

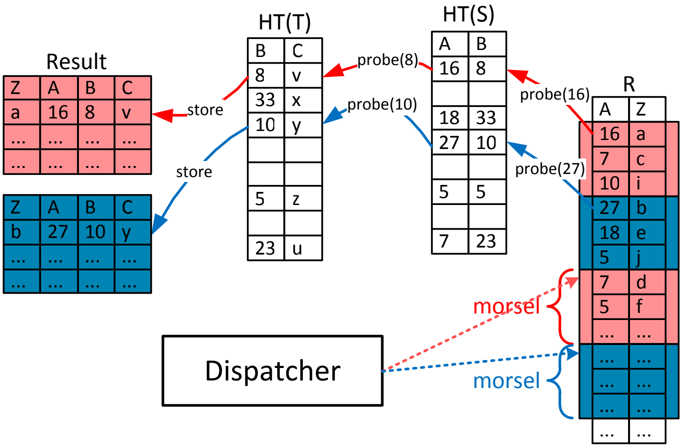
- static work partitioning can cause load imbalances (large problem with many threads)
- degree of parallelism cannot easily be changed mid-query (workload changes)
- overhead:
  - usually implemented using more threads than CPU cores (context switching)
  - hash re-partitioning often does not pay off
  - exchange operators create additional copies of the tuples

Parallel Query Engine

- alternative to Exchange Operators: parallelize operators themselves
- requires synchronization of shared data structures (e.g., hash tables)
- allows for more flexibility in designing parallel algorithms for relational operators

Morsel-Driven Query Execution

- break input into constant-sized work units (“morsels”)
- dispatcher assigns morsels to worker threads
- $\#$ worker threads $=$ $\#$ hardware threads



Dynamic Scheduling

- the total runtime of a query is the runtime of the slowest thread/core/machine
- when dozens of cores are used, often a single straggler is much slower than the others (e.g., due to other processes in the system or non-uniform data distributions)
- solution: don't partition input data at the beginning, but use dynamic work stealing:
  - synchronized queue of small jobs
  - threads grab work from queue
  - the `parallel_for` construct can provide a high-level interface

Parallel In-Memory Hash Join

1. build phase:
   1.1 each thread scans part of the input and materializes the tuple
   1.2 create table of pointers of appropriate size (tuple count sum of all threads)
   1.3 scan materialized input and add pointers from array to materialized tuples using atomic instructions
2. probe phase: can probe the hash table in parallel without any synchronization (as long no marker is needed)