Efficient in-memory query execution using JIT compiling

Han-Gyu Park

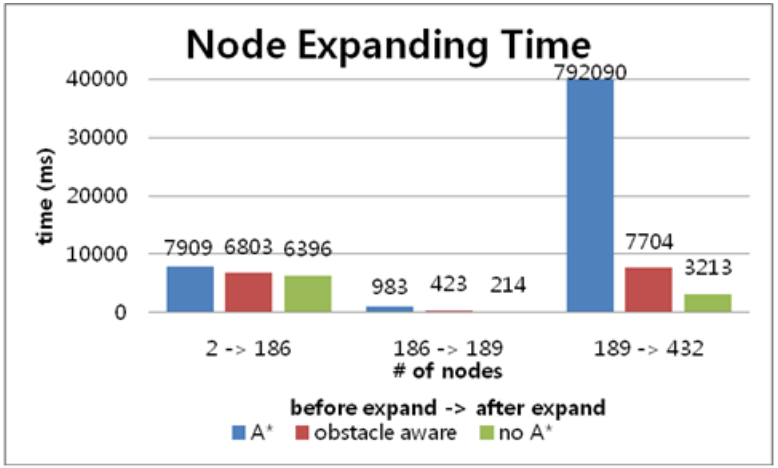
2012-11-16

CONTENTS

- **Introduction**
- **How DCX works**
- **Experiment**(~~purpose(at the beginning of this slide)~~, environment, ~~result, analysis & conclusion~~)
  - What I did
    - Test Query -> Query Plan -> Pseudo Code -> L Code
  - Test environment
  - Result & Analysis
    - Result of TPC-H Q3 in row store
    - Result of TPC-H Q1 in row/column store
    - Result of the VTune profile of TPC-H Q3
- **Conclusion**
- Reference

INTRODUCTION

- PlanViz edge routing (~8.3)
  - Improved performance up to 100 times



- **DCX for row store (8.6~now)**
  - Study on L codes and Neumann's paper for the first couple of weeks
  - Start converting codes for column store to ones for row store

INTRODUCTION(cont'd)

- Goal
  - To improve performance of query execution
  - Applying DCX concepts to row store

- **What is DCX?**
  - DCX stands for *data centric execution*
  - The main idea is to maximize code locality to avoid materialization
    - Significant growth of main memory
    - Query performance is more and more determined by the raw CPU costs
      - Avoiding materialization
      - Reducing branch prediction misses
  - DCX provides better code locality, therefore provides better performance

HOW DCX WORKS

- DCX

SQL

```
SELECT
    L_ORDERKEY,
    SUM(L_EXTENDEDPR
            L_DISCOUNT)),
    O_ORDERDATE,
    O_SHIPPRIORITY
FROM
    CUSTOMER,
    ORDERS,
    LINEITEM
WHERE
    C_MKTSEGMENT = '
    AND C_CUSTKEY = (
    AND L_ORDERKEY =
    AND O_ORDERDATE
    AND L_SHIPDATE >
GROUP BY
    L_ORDERKEY,
    O_ORDERDATE,
    O_SHIPPRIORITY
```

L codes

```
DROP SCHEMA LTEST_ROW CASCADE;
CREATE SCHEMA LTEST_ROW;

CREATE TYPE LTEST_ROW.RESULT AS TABLE("col0"char ( 1 ) not null,"col1" char ( 1 ) not null,"col2"decimal ( 12 , 2 ) not null,"col3"decimal ( 12 , 2 ) not
null,"col4"decimal ( 12 , 2 ) not null,"col5"decimal ( 12 , 2 ) not null,"col6"decimal ( 12 , 2 ) not null,"col7"decimal ( 12 , 2 ) not null,"col8"decimal ( 12 , 2 ) not
null,"col9"integer not null);
CREATE PROCEDURE LTEST_ROW.PLANFUNC(OUT result LTEST_ROW.RESULT) LANGUAGE LLANG READS SQL DATA AS
BEGIN
#pragma endlessloopprotection disable
typedef Table<FixedString <1>"col0", FixedString<1>"col1", Fixed8<2>"col2", Fixed8 <2>"col3", Fixed8<2>"col4", Fixed8<2>"col5", Fixed8<2>"col6", Fixed8<2>"col7",
Fixed8<2>"col8", Int32"col9"> ResultTable;

export Void main (ResultTable &result ) {
//---Access to a row table to get offsets
RowScan rowScan;
rowScan.access("SYSTEM","R_LINEITEM");
Int32 offset_L_DISCOUNT = rowScan.getOffset("L_DISCOUNT");
Int32 offset_L_EXTENDEDPRICE = rowScan.getOffset("L_EXTENDEDPRICE");
Int32 offset_L_LINESTATUS = rowScan.getOffset("L_LINESTATUS");
Int32 offset_L_QUANTITY = rowScan.getOffset("L_QUANTITY");
Int32 offset_L_RETURNFLAG = rowScan.getOffset("L_RETURNFLAG");
Int32 offset_L_SHIPDATE = rowScan.getOffset("L_SHIPDATE");
Int32 offset_L_TAX = rowScan.getOffset("L_TAX");

//---HashTable for aggregation
HashTable state_GroupBy;

//---Constant value for filtering
CSDate state_Constant;
state_Constant = CSDate("1998-09-02");

//---Declare result columns
Column<FixedString <1> > state_ResultColumn;
Column<FixedString <1> > state_ResultColumn1;
Column<Fixed8<2> > state_ResultColumn2;
Column<Fixed8<2> > state_ResultColumn3;
Column<Fixed8<2> > state_ResultColumn4;
Column<Fixed8<2> > state_ResultColumn5;
Column<Fixed8<2> > state_ResultColumn6;
Column<Fixed8<2> > state_ResultColumn7;
Column<Fixed8<2> > state_ResultColumn8;
Column<Int32 > state_ResultColumn9;
Size state_ResultCount;

state_ResultColumn = result.getColumn<FixedString <1> >(0z );
state_ResultColumn1 = result.getColumn<FixedString <1> >(1z);
state_ResultColumn2 = result.getColumn<Fixed8<2> >(2z );
state_ResultColumn3 = result.getColumn<Fixed8<2> >(3z );
state_ResultColumn4 = result.getColumn<Fixed8<2> >(4z );
state_ResultColumn5 = result.getColumn<Fixed8<2> >(5z );
state_ResultColumn6 = result.getColumn<Fixed8<2> >(6z );
state_ResultColumn7 = result.getColumn<Fixed8<2> >(7z );
state_ResultColumn8 = result.getColumn<Fixed8<2> >(8z );
state_ResultColumn9 = result.getColumn<Int32 >(9z );
state_ResultCount = 0z;

//---Reset the hash table
state_GroupBy.reset();

//---For each tuples in lineitem
RawUInt64 tid = RawUInt64("0");
RawUInt64 tidLimit = RawUInt64 ("6001215");
while(tid < tidLimit) {
//---Filtering with shipdate: TODO: Comparison with Date type
CSDate attribute_L_SHIPDATE = CSDate(String(rowScan.getDateAt(offset_L_SHIPDATE)));
if(attribute_L_SHIPDATE < state_Constant) {
//---Get all the other attributes: TODO: impl. getDecAt()
Fixed8<2> attribute_L_DISCOUNT = Fixed8<2>(rowScan.getDecAt(offset_L_DISCOUNT));
Fixed8<2> attribute_L_EXTENDEDPRICE = Fixed8<2>(rowScan.getDecAt(offset_L_EXTENDEDPRICE));
FixedString<1> attribute_L_LINESTATUS = FixedString<1>(rowScan.getVarcharAt(offset_L_LINESTATUS, 1));
Fixed8<2> attribute_L_QUANTITY = Fixed8<2>(rowScan.getDecAt(offset_L_QUANTITY));
FixedString<1> attribute_L_RETURNFLAG = FixedString<1>(rowScan.getVarcharAt(offset_L_RETURNFLAG, 0));
Fixed8<2> attribute_L_TAX = Fixed8<2>(rowScan.getDecAt(offset_L_TAX));

//---Calculate result column values
Fixed8<2> value = attribute_L_EXTENDEDPRICE *( Fixed8 <2>("1.00")-attribute_L_DISCOUNT );
Fixed8<2> value10 = attribute_L_EXTENDEDPRICE *( Fixed8 <2>("1.00")-attribute_L_DISCOUNT ) *( Fixed8 <2>("1.00")+attribute_L_TAX );

//---Create hash
RawUInt64 hash = state_GroupBy.hash<FixedString<1> >(attribute_L_LINESTATUS , state_GroupBy.hash<FixedString<1> >(attribute_L_RETURNFLAG , RawUInt64
("0") ) );
HashTableIterator hashTableIterator = state_GroupBy.getBucket ( hash );

//---Fill in the hash table
while( true ) {
if ( ! hashTableIterator.isValid() ) {
RawUInt64 size = RawUInt64 ("8");
DataPointer ptr = state_GroupBy.insert( hash , size+RawUInt64 ("44") );
ptr.set<FixedString<1> >(attribute_L_RETURNFLAG );
ptr.set<FixedString<1> >(attribute_L_LINESTATUS );
DataPointer ptr11 = ptr;
ptr11.setAt<Fixed8<2> >(RawUInt64 ("0") , attribute_L_QUANTITY );
ptr11.setAt<Fixed8<2> >(RawUInt64 ("8") , attribute_L_EXTENDEDPRICE);
ptr11.setAt<Fixed8<2> >(RawUInt64 ("16") , value );
ptr11.setAt<Fixed8<2> >(RawUInt64 ("24") , value10 );
ptr11.setAt<Int32 >(RawUInt64 ("32") , 1 );
ptr11.setAt<Fixed8<2> >(RawUInt64 ("36") , attribute_L_DISCOUNT );
break;
}
if ( hashTableIterator.getHash() == hash ) {
DataPointer ptr12 = hashTableIterator.getData();
FixedString<1> tmp = ptr12.get<FixedString<1> >();
FixedString<1> tmp13 = ptr12.get<FixedString<1> >();
DataPointer ptr14 = ptr12;
if( attribute_L_RETURNFLAG == tmp && attribute_L_LINESTATUS == tmp13 ) {
ptr14.setAt<Fixed8<2> >(RawUInt64 ("0") , ptr14.getAt<Fixed8<2> >(RawUInt64 ("0") ) + attribute_L_QUANTITY );
ptr14.setAt<Fixed8<2> >(RawUInt64 ("8") , ptr14.getAt<Fixed8<2> >(RawUInt64 ("8") ) + attribute_L_EXTENDEDPRICE );
ptr14.setAt<Fixed8<2> >(RawUInt64 ("16") , ptr14.getAt<Fixed8<2> >(RawUInt64 ("16") ) + value );
ptr14.setAt<Fixed8<2> >(RawUInt64 ("24") , ptr14.getAt<Fixed8<2> >(RawUInt64 ("24") ) + value10 );
ptr14.setAt<Int32 >(RawUInt64 ("32") , ptr14.getAt<Int32 >(RawUInt64 ("32") ) + 1 );
ptr14.setAt<Fixed8<2> >(RawUInt64 ("36") , ptr14.getAt<Fixed8<2> >(RawUInt64 ("36") ) + attribute_L_DISCOUNT );
break;
}
hashTableIterator.next();
}
for( ; state_GroupBy.getSize() ; ) {
HashTableIterator hashTableIterator15 = state_GroupBy.getBucket(RawUInt64 ("0");
while( hashTableIterator15.isValid() ) {
DataPointer ptr16 = hashTableIterator15.getData();
FixedString<1> tmp17 = ptr16.get<FixedString<1> >();
FixedString<1> tmp18 = ptr16.get<FixedString<1> >();
Fixed8<2> tmp19 = ptr16.getAt<Fixed8<2> >(RawUInt64 ("0") );
Fixed8<2> tmp20 = ptr16.getAt<Fixed8<2> >(RawUInt64 ("8") );
Fixed8<2> tmp21 = ptr16.getAt<Fixed8<2> >(RawUInt64 ("16") );
Fixed8<2> tmp22 = ptr16.getAt<Fixed8<2> >(RawUInt64 ("24") );
Int32 tmp23 = ptr16.getAt<Int32 >(RawUInt64 ("32") );
Fixed8<2> tmp24 = ptr16.getAt<Fixed8<2> >(RawUInt64 ("36") );
state_ResultColumn[ state_ResultCount ] = tmp17;
state_ResultColumn1 [ state_ResultCount ] = tmp18;
state_ResultColumn2 [ state_ResultCount ] = tmp19;
state_ResultColumn3 [ state_ResultCount ] = tmp20;
state_ResultColumn4 [ state_ResultCount ] = tmp21;
state_ResultColumn5 [ state_ResultCount ] = tmp22;
state_ResultColumn6 [ state_ResultCount ] = tmp19/ Fixed8 <2>(tmp23 );
state_ResultColumn7 [ state_ResultCount ] = tmp20/ Fixed8 <2>(tmp23 );
state_ResultColumn8 [ state_ResultCount ] = tmp24/ Fixed8 <2>(tmp23 );
state_ResultColumn9 [ state_ResultCount ] = tmp23;
state_ResultCount = state_ResultCount+1z;
hashTableIterator15 = hashTableIterator15.getNext ( );
}
hashTableSlot = hashTableSlot+RawUInt64 ("1");
}
}
END;
CALL LTEST_ROW.PLANFUNC(?);
```

QUERIES FOR THE EXPERIMENT

- For the experiment, these SQL queries should be translated to codes that can be executed just-in-time
  - In this experiment, I used L code which can be JIT compiled with LLVM
  - SQL queries were translated manually

TPC-H Q1

```
SELECT
    L_RETURNFLAG,
    L_LINESTATUS,
    SUM(L_QUANTITY) AS SUM_QTY,
    SUM(L_EXTENDEDPRICE) AS
        SUM_BASE_PRICE,
    SUM(L_EXTENDEDPRICE*(1-
        L_DISCOUNT)) AS
        SUM_DISC_PRICE,
    SUM(L_EXTENDEDPRICE*(1-
        L_DISCOUNT)*(1+L_TAX)) AS
        SUM_CHARGE,
    AVG(L_QUANTITY) AS AVG_QTY,
    AVG(L_EXTENDEDPRICE) AS
        AVG_PRICE,
    AVG(L_DISCOUNT) AS AVG_DISC,
    COUNT(*) AS COUNT_ORDER
FROM
    LINEITEM
WHERE
    L_SHIPDATE <= '1998-09-02'
GROUP BY
    L_RETURNFLAG,
    L_LINESTATUS
```

TPC-H Q3

```
SELECT
    L_ORDERKEY,
    SUM(L_EXTENDEDPRICE*(1-
        L_DISCOUNT)) AS REVENUE,
    O_ORDERDATE,
    O_SHIPPRIORITY
FROM
    CUSTOMER,
    ORDERS,
    LINEITEM
WHERE
    C_MKTSEGMENT = 'BUILDING'
    AND C_CUSTKEY = O_CUSTKEY
    AND L_ORDERKEY = O_ORDERKEY
    AND O_ORDERDATE < '1995-03-15'
    AND L_SHIPDATE > '1995-03-15'
GROUP BY
    L_ORDERKEY,
    O_ORDERDATE,
    O_SHIPPRIORITY
```

PSEUDO CODE

- **Maximize code locality to boost performance**
  - L codes are manually written
    - According to pseudo codes which maximizes code locality
    - Below is a pseudo code of TPC-H query 3

Pseudo code for TPC-H query 3

```
initialize memory of ⋈custkey, ⋈orderkey, and Γ
for each tuple t in CUSTOMER
    if t.MKTSEGMENT = 'BUILDING'
        materialize t in hash table of ⋈custkey
for each tuple t in LINEITME
    if t.SHIPDATE > '1995-03-15'
        materialize t in hash table of ⋈orderkey
for each tuple t3 in ORDERS
    if t3.ORDERDATE < '1995-03-15'
        for each match t2 in ⋈custkey[t3.CUSTKEY]
         for each match t1 in ⋈orderkey[t2.ORDERKEY]
           aggregate t1 in hash table of Γ
```

AN EXAMPLE OF L CODE

L code for highlighted part (TPC-H query 3)

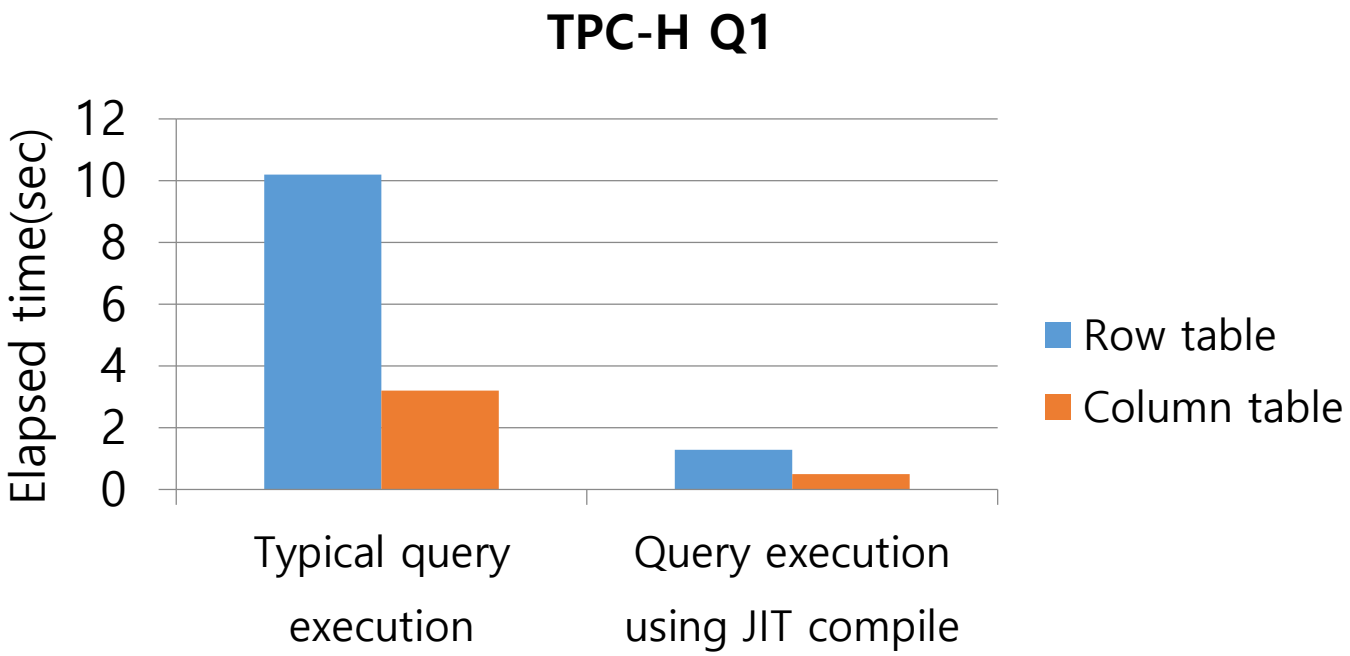
```
RawUInt64 customer_tid = RawUInt64("0");
RawUInt64 customer_tidLimit = RawUInt64("150000");
while( customer_tid < customer_tidLimit) {
  String attribute_C_MKTSEGMENT;
  rowScan_CUSTOMER.getVarcharAt(offset_C_MKTSEGMENT, idx_C_MKTSEGMENT,
    attribute_C_MKTSEGMENT);
  if(attribute_C_MKTSEGMENT == state_Constant_C_MKTSEGMENT) {
    Int32 attribute_C_CUSTKEY = rowScan_CUSTOMER.getIntAt(offset_C_CUSTKEY);

    RawUInt64 hash_Join1 = state_Join1.hash<Int32 >(attribute_C_CUSTKEY , RawUInt64 ("0"));
    HashTableIterator hash_Join1_iterator = state_Join1.getBucket ( hash_Join1 );

    while( true ) {
      if ( ! hash_Join1_iterator.isValid( ) ) {
        RawUInt64 size = RawUInt64 ("4");
        DataPointer ptr = state_Join1.insert( hash_Join1 , size+RawUInt64 ("0") );
        ptr.set<Int32 >(attribute_C_CUSTKEY );
        break;}
      if ( hash_Join1_iterator.getHash() == hash_Join1 ) {
        //--Add to hash
        RawUInt64 size = RawUInt64 ("4");
        DataPointer ptr = state_Join1.insert( hash_Join1 , size+RawUInt64 ("0") );
        ptr.set<Int32 >(attribute_C_CUSTKEY );
        break;}
      hash_Join1_iterator = hash_Join1_iterator.getNext();}}
  customer_tid = customer_tid + RawUInt64("1");
  rowScan_CUSTOMER.advance();}
```

- **Manually written L code**

EXPERIEMNT ENVIRONMENT

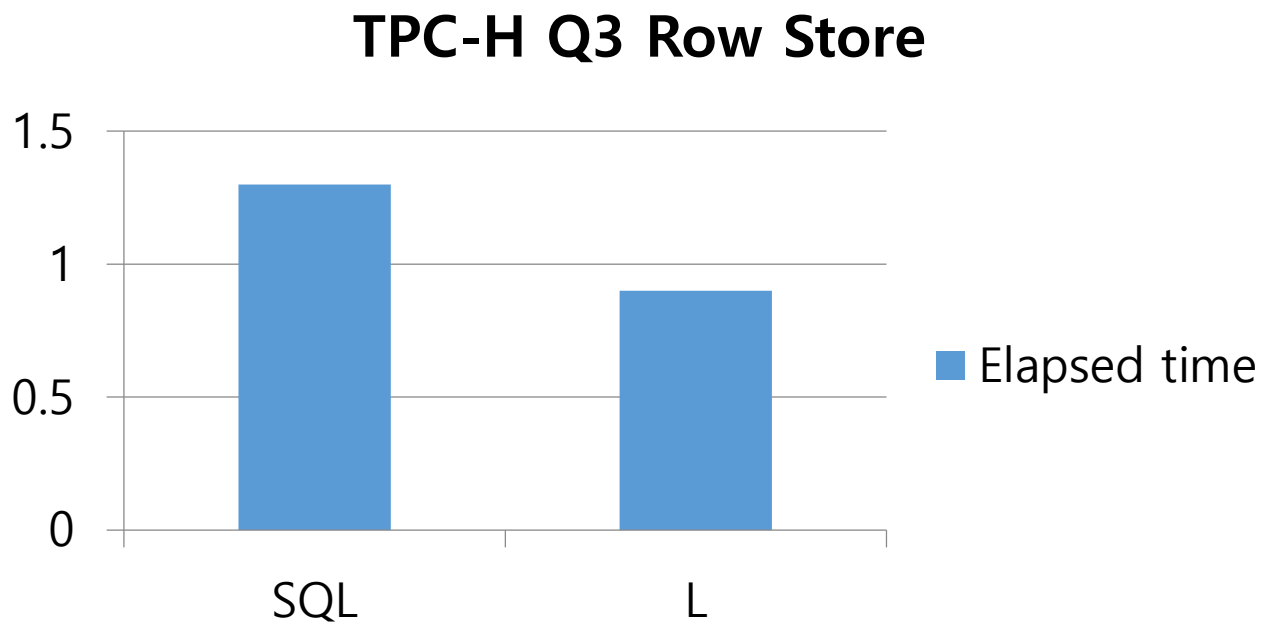
- **All experiments was conducted with**
  - **seltera machine**
    - CPU: Intel(R) Xeon(R) CPU E7- 4870 @ 2.40GHz / 40 core / LLC size = 30MB
    - Memory: 1TB
    - OS: SLES 11 SP1
  - **Single thread**
  - **Column/row table set for each experiment**

RESULT(1)



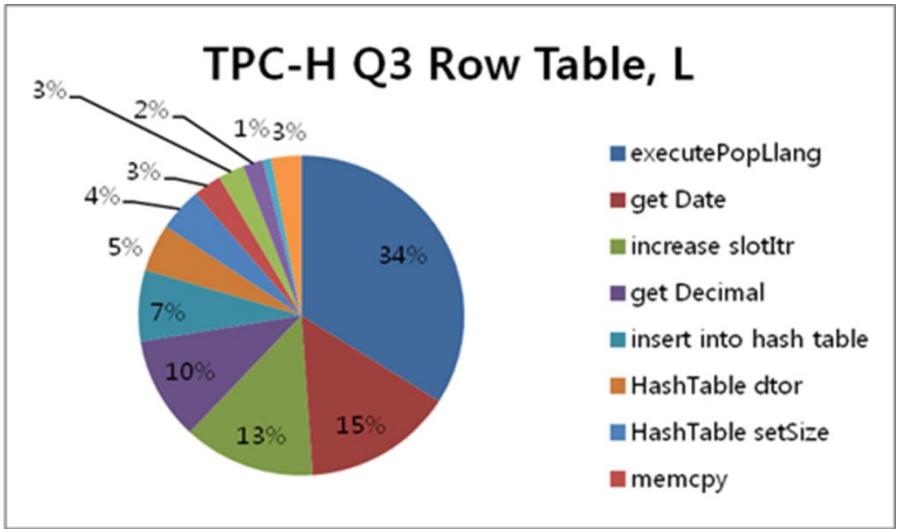
- Wrote TPC-H Q1 in L codes for both row and column stores
- Improvement for both types
  - We now know that it is generally applicable concept
- No parallelism

RESULT(2)



- Better improvement than Q1 because of more pipeline breakers
  - more joins
  - aggregation
- No parallelism

RESULT(3)



- Categorizing the VTune profile result:
  - L language part costs 34%
  - Get Date, Decimal Value costs 25%
  - Increasing slotItr costs 13%
  - Hash Table operations costs 16%

Reference

- T. Neumann. Efficiently compiling efficient query plans for modern hardware. In VLDB, 2011.