

Consider the following Java-JDT plugin name in German:

Consider the following Java-JDT plugin name in German:

BOSTON UNIVERSITY
COLLEGE OF ENGINEERING

Dissertation

A BU THESIS LATEX TEMPLATE

by

JOE CANDIDATE

B.S., Some University, 2010
M.S., Another University, 2012

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2015

© 2015 by
JOE CANDIDATE
All rights reserved

Approved by

First Reader

First M. Last, PhD
Professor of Electrical and Computer Engineering

Second Reader

First M. Last
Associate Professor of ...

Third Reader

First M. Last
Assistant Professor of ...

*Facilis descensus Averni;
Noctes atque dies patet atri janua Ditis;
Sed revocare gradum, superasque evadere ad auras,
Hoc opus, hic labor est.* *Virgil (from Don's thesis!)*

Acknowledgments

Here go all your acknowledgments. You know, your advisor, funding agency, lab mates, etc., and of course your family.

As for me, I would like to thank Jonathan Polimeni for cleaning up old LaTeX style files and templates so that Engineering students would not have to suffer typesetting dissertations in MS Word. Also, I would like to thank IDS/ISS group (ECE) and CV/CNS lab graduates for their contributions and tweaks to this scheme over the years (after many frustrations when preparing their final document for BU library). In particular, I would like to thank Limor Martin who has helped with the transition to PDF-only dissertation format (no more printing hardcopies – hooray !!!)

The stylistic and aesthetic conventions implemented in this LaTeX thesis/dissertation format would not have been possible without the help from Brendan McDermot of Mugar library and Martha Wellman of CAS.

Finally, credit is due to Stephen Gildea for the MIT style file off which this current version is based, and Paolo Gaudiano for porting the MIT style to one compatible with BU requirements.

Janusz Konrad

Professor

ECE Department

A BU THESIS LATEX TEMPLATE

JOE CANDIDATE

Boston University, College of Engineering, 2015

Major Professors: First M. Last, PhD

Professor of Electrical and Computer Engineering

Secondary appointment

First M. Last, PhD

Professor of Computer Science

ABSTRACT

Have you ever wondered why this is called an abstract? Weird thing is that its legal to cite the abstract of a dissertation alone, apart from the rest of the manuscript.

Contents

1	<i>Introduction</i>	1
1.1	<i>Motivation</i>	1
1.2	<i>Problem at hand</i>	1
1.3	<i>Structure of thesis</i>	1
1.4	<i>Conclusion</i>	1
2	<i>Related Work</i>	2
2.1	<i>Introduction, Query optimization</i>	2
2.2	<i>Converting SQL queries to parse trees</i>	2
2.3	<i>Relational algebra</i>	3
2.3.1	<i>Select operator σ</i>	4
2.3.2	<i>Projection operator π</i>	5
2.3.3	<i>Duplicate Elimination operator δ</i>	5
2.3.4	<i>Aggregation operator γ</i>	6
2.4	<i>Converting Parse trees into logical expression</i>	6
2.5	<i>Explain difficulties/ Time complexity</i>	8
2.5.1	<i>Estimating size and cost</i>	9
2.5.2	<i>Estimation of Projection</i>	9
2.5.3	<i>Estimation of Selection</i>	10
2.5.4	<i>Estimation of Join, single attribute</i>	10
2.5.5	<i>Estimation of Join, multiple attribute</i>	11
2.5.6	<i>Multiple Joins</i>	12

2.5.7	<i>Union</i>	12
2.5.8	<i>Intersection</i>	12
2.5.9	<i>Difference</i>	13
2.5.10	<i>Duplicate Elimination</i>	13
2.5.11	<i>Grouping and Aggregation</i>	13
2.6	<i>Other tools</i>	13
2.6.1	<i>Histogram</i>	13
2.6.2	<i>Heuristics</i>	14
2.7	<i>Enumeration Methods</i>	15
2.7.1	<i>Heuristic Selection</i>	16
2.7.2	<i>Branch-and-Bound</i>	16
2.7.3	<i>Hill Climbing</i>	17
2.7.4	<i>Selinger-Style Optimization</i>	17
2.8	<i>Join Order</i>	18
2.8.1	<i>Join Trees</i>	18
2.8.2	<i>DP to decide join order</i>	19
2.8.3	<i>Greedy algorithm for join order</i>	20
2.9	<i>Physical Query Plan</i>	20
2.9.1	<i>Choosing a Selection Method</i>	21
2.9.2	<i>Choosing a Join Method</i>	22
2.9.3	<i>Pipelining Versus Materialization</i>	23
2.9.4	<i>Pipelining Unary Operations</i>	23
2.9.5	<i>Pipelining Binary Operations</i>	24
2.9.6	<i>Notation for Physical Query Plans</i>	24
2.9.7	<i>Ordering of Physical Operations</i>	24
2.10	<i>Introduction to Data Streams</i>	25

2.11	<i>Data stream windowing</i>	25
2.12	<i>Query Processing of data streams(Combine the DBMS and DSMS)</i>	25
2.13	<i>Challenges of query optimization on data streams</i>	25
2.14	<i>Conclusion and discussion</i>	25
3	<i>Stream Optimization</i>	26
3.1	<i>Query Optimization of Data Streams</i>	26
4	<i>Implementation</i>	27
4.1	<i>Mathematics</i>	27
5	<i>Evaluation</i>	28
5.1	<i>Measures used</i>	28
6	<i>Conclusion and Further work</i>	29
6.1	<i>Conclusion</i>	29
A	<i>Proof of xyz</i>	30
	<i>Curriculum Vitae</i>	31

List of Tables

List of Figures

List of Abbreviations

The list below must be in alphabetical order as per BU library instructions or it will be returned to you for re-ordering.

<i>CAD</i>	<i>Computer-Aided Design</i>
<i>CO</i>	<i>Cytochrome Oxidase</i>
<i>DOG</i>	<i>Difference Of Gaussian (distributions)</i>
<i>FWHM</i>	<i>Full-Width at Half Maximum</i>
<i>LGN</i>	<i>Lateral Geniculate Nucleus</i>
<i>ODC</i>	<i>Ocular Dominance Column</i>
<i>PDF</i>	<i>Probability Distribution Function</i>
\mathbb{R}^2	<i>the Real plane</i>

Chapter 1

Introduction

1.1 Motivation

1.2 Problem at hand

Hello

1.3 Structure of thesis

Works

1.4 Conclusion

The next chapter gives an in-depth view of the pipeline used by the current state of art technology for query optimization in traditional data bases including the mathematical knowledge for simplification and the overall framework. The next chapter also introduces the reader to data stream and how data bases are used for them called DSMS and showcases an approach to optimize queries on data streams for the problem discussed above. The following chapter list out the details of implementation, challenges face, evaluation methods used, benchmark test case timings, followed by a summary of the paper.

Chapter 2

Related Work

2.1 Introduction, Query optimization

A database can be thought of as a list of tables, where in each table itself can be considered as a list of data points ordered initially in the sequence they are entered.

There are various tools which can be used to connect to a database, here we focus on structured query languages(SQL). A simple SQL query looks like this

```
1  SELECT column_name_1 , column_name_2
2  FROM table_name
3  WHERE condition
```

This query is essentially asking to display the 2 columns from the table where the condition given is satisfied. This to particular query might be looking simple, but if the condition introduced is a complex one or if the table from which we need to return the output is complex, the question of how to execute the query optimally becomes difficult to answer.

2.2 Converting SQL queries to parse trees

This step has several functions.

If a "view" is used in the query as a relation, then each instance has to be replaced by the parse tree.

The preprocessor also has to conduct semaContainment of Value Sets. If Y is an attribute appearing in several rela tions, then each relation chooses its values from

the front of a fixed list of values y_1, y_2, y_3 , and has all the values in that prefix. As a consequence, if R and S are two relations with an attribute Y , and $V(R, Y) \subseteq V(S, Y)$, then every Y -value of R will be a Y -value of S .
 syntic checking, that is, check if relations used exist, check for ambiguity, and type checking. If a parse tree passes the preprocessing then it is said to be **valid**. We don't describe the exact grammar for the conversion to the parse tree. In these parse trees, there are 2 types of nodes, one the atoms, which are essentially keywords in SQL, operators, constants and attributes. The second is Syntactic categories, these are names for families of subqueries in triangular brackets. Each of the syntactic category has unique expansion into atoms and further syntactic categories.

2.3 Relational algebra

As we saw above, order of operations matters, if the order of operations is not thoughtout and done blindly alot of redundant steps are executed and memory is moved around unnecessarily. There are few ways to atleast look and analyse the operations and how they can be simplified.

Let R, S be relations. Some simple laws, associativity and commutativity can easily be verified:-

- $R \times S = S \times R$
- $(R \times S) \times T = R \times (S \times T)$
- $R \bowtie S = S \bowtie R$

- $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- $R \cup S = S \cup R$
- $(R \cup S) \cup T = R \cup (S \cup T)$
- $R \cap S = S \cap R$
- $(R \cap S) \cap T = R \cap (S \cap T)$

When applying associative law on relations, need to be careful whether the conditions actually makes sense after the order is changed.

While the above identities work on both sets and bags(bags allow for repeatition). To show that laws for sets and bags do differ an easy way is to consider the distributive property.

$$A \cap_S (B \cup_S C) = (A \cap_S B) \cup_S (A \cap_S C)$$

$$A \cap_B (B \cup_B C) \neq (A \cap_B B) \cup_B (A \cap_B C)$$

We can simply show it with an example. Let $A = \{t\}, B = \{t\}, C = \{t\}$. The LHS comes to be $\{t\}$, whereas RHS is $\{t, t\}$

2.3.1 Select operator σ

First we start with simple properties of the σ operator. Need to be careful about the attributes used in the select operator condition when pushing it down.

- $\sigma_{C_1 \wedge C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$
- $\sigma_{C_1 \vee C_2}(R) = (\sigma_{C_1}(R)) \cup_S (\sigma_{C_2}(R))$
- $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
- $\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S) = \sigma_C(R) - S$

- $\sigma_C(R \times S) = \sigma_C(R) \times S$
- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
- $\sigma_C(R \bowtie_D S) = \sigma_C(R) \bowtie_D S$
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

2.3.2 Projection operator π

While for the Select operator(σ) the identities were quite straight forward with not many things to consider, the identities for Projection operator (π) are bit more involved.

- $\pi_L(R \bowtie S) = \pi_L(\pi_M(R) \bowtie \pi_N(S))$, where M, N are attributes required for the join or they are inputs to the projection.
- $\pi_L(R \bowtie_D S) = \pi_L(\pi_M(R) \bowtie_D \pi_N(S))$, similar to above identity/ law.
- $\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$
- $\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S)$
- $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$

2.3.3 Duplicate Elimination operator δ

The δ operator eliminates duplicates from bags.

- $\delta(R) = R$, if R does not have any duplicates.
- $\delta(R \times S) = \delta(R) \times \delta(S)$
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
- $\delta(R \bowtie_D S) = \delta(R) \bowtie_D \delta(S)$

- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$
- $\delta(R \cap_B S) = \delta(R) \cap_B S$

2.3.4 Aggregation operator γ

It is difficult to give identities for the aggregation operator, like done for the above operators. This is mostly due to how the details of how the aggregation operator is used.

- $\sigma(\gamma_L(R)) = \gamma_L(R)$
- $\gamma_L(R) = \gamma_L(\pi_M(R))$, where M must at least contain the attributed used in L .

2.4 Converting Parse trees into logical expression

Till now, the only SQL related information presented is how to convert a Query into the parse tree, which is grammar dependent. Given the parse tree, need to substitute nodes by operators seen above, later this expression is optimized to be later converted to a physical query plan.

Now to convert the parse tree into the logical expression. First, look at the transformation of select-from-where statement.

- $\langle \text{Query} \rangle \rightarrow \langle \text{SFW} \rangle$.
- $\langle \text{SFW} \rangle \rightarrow \text{SELECT } \langle \text{SelList} \rangle \text{ FROM } \langle \text{FromList} \rangle \text{ WHERE } \langle \text{Condition} \rangle$.
- $\langle \text{SelList} \rangle \rightarrow \pi_L$, where L is the list of attributes in $\langle \text{SelList} \rangle$.
- $\langle \text{Condition} \rangle \rightarrow \sigma_C$, where C is the equivalent of $\langle \text{Condition} \rangle$

While there is nothign inherently wrong about the last statement, need to consider the case where $\langle \text{Condition} \rangle$ involves subqueries. A simple explanation about why it

isn't allowed is a convention normally the subscript has to be a boolean condition, and if it was allowed otherwise, it would be a very expensive operation, as the subscript in the select operator has to be evaluated at every element of the argument relation. This shows the redundancy of it. While if this is allowed, it can be simplified and made efficient, but has to be done on a case by case basis with the use of \bowtie, \times functions.

Overall it is a good idea to not use subquerying and rather using joins.

At this point, by making the substitutions mentioned above and using the algebraic identities, we obtain a starting logical query plan. The query has to be transformed into a query which the compiler believes to be the cheapest or the optimal. But, a thing which further complicates the process is the join order.

With the current knowledge, few optimizing rules are evident.

- **Selection repositioning**, Selections should be pushed down as much as possible, but sometimes, might need to take the selection operator a level up first.
- Pushing projections down the parse tree, being careful with the new projections made in the process.
- Duplicate removal needs to be repositioned.
- σ combined with \times below can result in equijoin, which is much more efficient.

Normally, parsers only have nodes with 0,1,2 children, which corresponds to unary and binary operators. But many of the operators(natural join, union, and intersection) are commutative and associative, so it helps combine them on a single level to provide the opportunity to prioritize which ones are done first. To do this step, the following guidelines are sufficient.

- We must replace the natural joins with theta-joins that equate the attributes of the same name.

- *We must add a projection to eliminate duplicate copies of attributes involved in a natural join that has become a theta-join*
- *The theta-join conditions must be associative.*

2.5 Explain difficulties/ Time complexity

Currently, we have taken a query as an input and converted it into a logical plan, and then applied more transformation using relational algebra to make the optimal query plan. The next step is to convert it into a physical query plan which can be executed. To complete this step, need to compare the cost of various physical query plan derived from the logical query plan. This plan with the least cost is then passed query execution engine and executed. When trying to select a query plan, need to select:-

- *An order and grouping for associative-and-commutative operations like joins, unions, and intersections.*
- *An algorithm for each operator in the logical plan, for instance, deciding whether a nested-loop join or a hash-join should be used.*
- *Additional operators - scanning, sorting, and so on that are needed for the physical plan but that were not present explicitly in the logical plan.*
- *The way in which arguments are passed from one operator to the next, for instance, by storing the intermediate result on disk or by using iterators and passing an argument one tuple or one main-memory buffer at a time.*

But after selecting these for a physical plan, one obvious way to calculate the time is actually executing the plan to see the cost, but that way essentially every plan is carried out. That is expensive and redundant. Is there a better way?

2.5.1 Estimating size and cost

$B(R) :=$ is the number of blocks needed to hold all the tuples of relation R .

$T(R) :=$ is the number of tuples of relation R .

$V(R, a) :=$ is the value count for attribute a of relation R , that is, the number of distinct values relation R has in attribute a .

$V(R, [a_1, a_2, \dots, a_n]) :=$ is the number of distinct values R has when all of attributes a_1, a_2, \dots, a_n are considered together, that is, the number of tuples in $\delta(\pi_{a_1, a_2, \dots, a_n}(R))$

Need to remember that physical plan is selected to minimize the estimated cost of evaluating the query and that intermidate/ temporary relations calculated will also incur a cost on the system. So need to take them into account as well. Over all the estimation method should pass the following sanity check:-

- Give accurate estimates. No matter what method is used for executing query plans.
- Are easy to compute.
- Are logically consistent; that is, the size estimate for an intermediate relation should not depend on how that relation is computed. For instance, the size estimate for a join of several relations should not depend on the order in which we join the relations.

But there is no agreed upon method to do this, but the goal is to help select a query plan and not to find the exact minimum. So even if the estimated cost is wrong, but if it is wrong similarly then we will still have the least costing plan.

2.5.2 Estimation of Projection

The projection operator is a bag operation, so it does not reduce the number of tuples, only reduces the size of each tuple.

$$T(R) = T(\pi(R))$$

$$B(R) \leq B(\pi(R))$$

2.5.3 Estimation of Selection

Let $S = \sigma_{A=c}(R)$, here A is an attribute of R and c is a constant

$T(S) = \frac{T(R)}{V(R,A)}$, is it important to note that this is an estimate and not the actual value, this will be the actual value if all the attributes in A have equal occurrence. An even better estimate can be obtained if the DBMS stores a statistic known as histogram.

The above calculation was easy because of the equality. if $S = \sigma_{A < c}(R)$, we simply estimate $T(S) = \frac{T(R)}{3}$

Now if $S = \sigma_{A \neq c}(R)$, can take $T(S) = T(R)$ or $T(S) = T(R) - \frac{T(R)}{V(R,A)}$

If $S = \sigma_{A=c_1 \vee c_2}$, take them to be independent conditions.

2.5.4 Estimation of Join, single attribute

For natural joins, we assume, the natural join of two relations involves only the equality of two attributes. That is, we study the join $R(X, Y) \bowtie S(Y, Z)$, but initially we assume that Y is a single attribute although X and Z can represent any set of attributes. But it is hard to find a good estimate as $T(R \bowtie S) \in [0, T(R) * T(S)]$, to help with this two assumptions are made.

- **Containment of Value Sets** If Y is an attribute appearing in several relations, then each relation chooses its values from the front of a fixed list of values y_1, y_2, y_3, \dots and has all the values in that prefix. As a consequence, if R and S are two relations with an attribute Y , and $V(R, Y) \leq V(S, Y)$, then every Y -value of R will be a Y -value of S .
- **Preservation of Value Sets** If we join a relation R with another relation, then an attribute A that is not a join attribute (i.e., not present in both relations)

does not lose values from its set of possible values. More precisely, if A is an attribute of R but not of S , then $V(R \bowtie S, A) = V(R, A)$. Note that the order of joining R and S is not important, so we could just as well have said that $V(S \bowtie R, A) = V(R, A)$.

Using the above assumptions we can claim,

$$T(R \bowtie S) = \frac{T(R) * T(S)}{\max(V(R, Y), V(S, Y))}$$

Let $V(R, Y) \leq V(S, Y)$, then every tuple t of R has a chance of $\frac{1}{V(S, Y)}$ of joining with a given tuple of S . Since there are $T(S)$ tuples in S , the expected number of tuples that t joins with is $\frac{T(S)}{V(S, Y)}$. As there are $T(R)$ tuples of R , the estimated size of $R \bowtie S$ is $\frac{T(R) * T(S)}{V(S, Y)}$, if it was $V(R, Y) \geq V(S, Y)$, then $\frac{T(R) * T(S)}{V(R, Y)}$

Guidelines for other type of joins:-

- The number of tuples in the result of an equijoin can be computed exactly as for a natural join, after accounting for the change in variable names.
- Other theta-joins can be estimated as if they were a selection following a product.

2.5.5 Estimation of Join, multiple attribute

Now we assume Y in $R(X, Y) \bowtie S(Y, Z)$ represents multiple attributes. Say for example, $R(X, y_1, y_2) \bowtie S(y_1, y_2, Z)$. We again do a probability calculation.

Let $r \in R, s \in S$ be a tuples, the probability that r, s agree on y_1 is $\frac{1}{\max(V(R, y_1), V(S, y_1))}$, similarly for y_2 . So we get the expected value to be

$$\frac{T(R) * T(S)}{\max(V(R, y_1), V(S, y_1)) * \max(V(R, y_2), V(S, y_2))}$$

From this the pattern is clear, need to divide $T(R) * T(S)$ by maximum of $V(R, y), V(S, y)$ for each attribute.

But this is only the calculation for $T(R \bowtie S)$, need to calculate for $B(R \bowtie S)$ as well.

2.5.6 Multiple Joins

For this case we work with $S = R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$

Here we have to make use of the containment assumption. Say the attribute A appears in k of R_i 's and the values corresponding to $V(R_i, A)$ are $v_1 \leq v_2 \leq \dots \leq v_k$, need to find the probability that tuples agree on A .

*Consider the tuple t_1 chosen from the relation that has the smallest number of A -values, v_1 . By the containment assumption, each of these v_1 values is among the A -values found in the other relations that have attribute A . Consider the relation that has V_i values in attribute A . Its selected tuple t_i has probability $\frac{1}{v_1}$ of agreeing with t_1 on A . Since this claim is true for all $i \in \{2, 3, \dots, k\}$, the probability that all k tuples agree on A is the product $\frac{1}{v_2 * v_3 * \dots * V_k}$. This analysis gives us the rule for estimating the size of any join.*

Start with the product of the number of tuples in each relation. Then for each attribute A appearing at least twice, divide by all but the least of the $V(R, A)$'s

2.5.7 Union

If U_B is used, it is the sum of the two individually.

If U_S is used, then number of tuples range from the max of two, to their sum. So take mean of the range.

2.5.8 Intersection

The number of tuples here ranges from 0 to the minimum of the two (in case of set intersection), so again can take the mean of this range.

2.5.9 Difference

The range for $R - S$ is $[T(R) - T(S), T(R)]$, so again mean of the range.

2.5.10 Duplicate Elimination

The range for $\delta(R)$ is $[1, T(R)]$, so again can take the mean, there can be other estimates as well.

A nice compromise is $\min(\frac{T(R)}{2}, \prod V(R, a_i))$

2.5.11 Grouping and Aggregation

Same as duplicate.

2.6 Other tools

We assume that the "cost" of evaluating an expression is approximated well by the number of disk I/O's performed. The number of disk I/O's, in turn, is influenced by:

- *The particular logical operators chosen to implement the query, a matter decided when we choose the logical query plan.*
- *The sizes of intermediate results.*
- *The physical operators used to implement logical operators, e.g., the choice of a one-pass or two-pass join, or the choice to sort or not sort a given relation.*
- *The ordering of similar operations, especially joins.*
- *The method of passing arguments from one physical operator to the next.*

2.6.1 Histogram

In the earlier section the statistics were heavily used in calculations. Another statistic that can be stored is the histogram.

If $V(R, A)$ is not too large, then the histogram may consist of the number (or fraction) of the tuples having each of the values of attribute A . If there are a great many values of this attribute, then only the most frequent values may be recorded individually, while other values are counted in groups.

Equal-width select 2 parameters, w the width and v_0 a beginning point of a column in the histogram, which will initially be the considered the lower bound and if an even lower value is noticed, make a smaller column as well and update the lower bound.

Equal-height These are the common "percentiles". A percentile $p, 2p$, so on.

Most-frequent-values List the most common values and their numbers of occurrences. This information may be provided along with a count of occurrences for all the other values as a group, or we may record frequent values in addition to an equal-width or equal-height histogram for the other values.

2.6.2 Heuristics

One important use of cost estimates for queries or subqueries is in the application of heuristic transformations of the query.

Heuristics applied independent of cost estimates can be expected almost certainly to improve the cost of a logical query plan

However, there are other points in the query optimization process where estimating the cost both before and after a transformation will allow us to apply a transformation where it appears to reduce cost and avoid the transformation otherwise. In particular, when the preferred logical query plan is being generated, we may consider a number of optional transformations and the costs before and after. Because we are estimating the cost of a logical query plan, so we have not yet made decisions about the physical operators that will be used to implement the operators of relational algebra, our cost estimate cannot be based on disk I / O's. Rather, we estimate the sizes of all intermediate results their sum is our heuristic estimate for the cost of the entire logical

plan.

2.7 Enumeration Methods

The naive method of find the least costing plan is enumerating all the possible plans and calculating the cost for them and then selecting the minimum one.

Top-Down *Here, we work down the tree of the logical query plan from the root. For each possible implementation of the operation at the root, we consider each possible way to evaluate its argument(s) , and compute the cost of each combination, taking the best.*

Bottom-up *For each subexpression of the logical-query-plan tree, we compute the costs of all possible ways to compute that subexpression. The possibilities and costs for a subexpression E are computed by considering the options for the subexpressions for E , and combining them in all possible ways with implementations for the root operator of E .*

*There is actually not much difference between the two approaches in their broadest interpretations, since either way, all possible combinations of ways to implement each operator in the query tree are considered. When limiting the search, a top-down approach may allow us to eliminate certain options that could not be eliminated bottom-up. However, bottom-up strategies that limit choices effectively have also been developed. there is an apparent simplification of the bottom-up method, where we consider only the best plan for each subexpression when we compute the plans for a larger subexpression. This approach, called **dynamic programming**, is not guaranteed to yield the best plan, although often it does. The approach called Selinger-style (or System-R-style) optimization exploits additional properties that some of the plans for a subexpression may have, in order to produce optimal overall plans from plans that are not optimal for certain subexpressions.*

2.7.1 Heuristic Selection

One option is to use the same approach to selecting a physical plan that is generally used for selecting a logical plan: make a sequence of choices based on heuristics. Few common ones are:-

- *If the logical plan calls for a selection $\sigma_{A=c}(R)$, and stored relation R has an index on attribute A , then perform an index-scan to obtain only the tuples of R with A -value equal to c .*
- *if the selection involves one condition like $A = c$ above, and other conditions as well, we can implement the selection by an index scan followed by a further selection on the tuples, which we shall represent by the physical operator filter.*
- *If an argument of a join has an index on the join attribute(s), then use an index-join with that relation in the inner loop.*
- *If one argument of a join is sorted on the join attribute(s) , then prefer a sort-join to a hash-join, although not necessarily to an index-join if one is possible.*
- *When computing the union or intersection of three or more relations, group the smallest relations first.*

2.7.2 Branch-and-Bound

This approach, often used in practice, begins by using heuristics to find a good physical plan for the entire logical query plan. Let the cost of this plan be C . Then as we consider other plans for subqueries, we can eliminate any plan for a sub query that has a cost greater than C , since that plan for the sub query could not possibly participate in a plan for the complete query that is better than what we already know. Likewise, if we construct a plan for the complete query that has cost less than C , we replace C

by the cost of this better plan in subsequent exploration of the space of physical query plans. An important advantage of this approach is that we can choose when to cut off the search and take the best plan found so far. For instance, if the cost C is small, then even if there are much better plans to be found, the time spent finding them may exceed C , so it does not make sense to continue the search. However, if C is large, then investing time in the hope of finding a faster plan is wise.

2.7.3 Hill Climbing

This approach, in which we really search for a "valley" in the space of physical plans and their costs, starts with a heuristically selected physical plan. We can then make small changes to the plan, e.g., replacing one method for an operator by another, or reordering joins by using the associative and/or commutative laws, to find "nearby" plans that have lower cost. When we find a plan such that no small modification yields a plan of lower cost, we make that plan our chosen physical query plan.

2.7.4 Selinger-Style Optimization

This approach improves upon the dynamic-programming approach by keeping for each subexpression not only the plan of least cost, but certain other plans that have higher cost, yet produce a result that is sorted in an order that may be useful higher up in the expression tree. Examples of such interesting orders are when the result of the subexpression is sorted on one of:

- The attribute(s) specified in a sort operator (τ) at the root.
- The grouping attribute(s) of a later group-by operator (γ).
- The join attribute(s) of a later join.

If we take the cost of a plan to be the sum of the sizes of the intermediate relations, then there appears to be no advantage to having an argument sorted. However, if we

use the more accurate measure, disk I/O's, as the cost, then the advantage of having an argument sorted becomes clear if we can use one of the sort-based algorithms and save the work of the first pass for the argument that is sorted already.

2.8 Join Order

Join takes in two argument, while the end result is independent of the order of the two arguments, the method used to compute the result maybe dependent on the order. Perhaps most important, the one-pass join reads one relation preferably the smaller into main memory, creating a structure such as a hash table to facilitate matching of tuples from the other relation. It then reads the other relation, one block at a time, to join its tuples with the tuples stored in memory.

For instance, suppose that when we select a physical plan we decide to use a one-pass join. Then we shall assume the left argument of the join is the smaller relation and store it in a main-memory data structure. This relation is called the build relation. The right argument of the join, called the probe relation, is read a block at a time and its tuples are matched in main memory with those of the build relation. Other join algorithms that distinguish between their arguments include:

- *Nested-loop join, where we assume the left argument is the relation of the outer loop.*
- *Index-join, where we assume the right argument has the index.*

2.8.1 Join Trees

When we have the join of two relations, we need to order the arguments. We shall conventionally select the one whose estimated size is the smaller as the left argument. Notice that the algorithms mentioned above – one-pass, nested loop, and indexed - each work best if the left argument is the smaller. More precisely, one-pass and nested-loop

joins each assign a special role to the smaller relation (build relation, or outer loop), and index-joins typically are reasonable choices only if one relation is small and the other has an index. It is quite common for there to be a significant and discernible difference in the sizes of arguments, because a query involving joins very often also involves a selection on at least one attribute, and that selection reduces the estimated size of one of the relations greatly.

*When we need to join more than 2 relations, the order in which they are joined can be represented by a binary tree, where each node has either 0 or 2 children. A tree where the right child always a leaf node is called a left deep tree, one can similarly define a right deep tree. Any other tree is will be called **bushy**. We will stick with left deep tree due to their interaction with various common join algorithms. This introduced limitation also helps to reduce search space.*

If one-pass joins are used, and the build relation is on the left, then the amount of memory needed at any one time tends to be smaller than if we used a right-deep tree or a bushy tree for the same relations.

If we use nested-loop joins, with the relation of the outer loop on the left, then we avoid constructing any intermediate relation more than once.

2.8.2 DP to decide join order

Suppose we need to calculate $R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$. For the DP algorithm construct a table with an entry for each subset of one or more of the n relations. In that table we put

- *the estimated size of the join of these relations.*
- *the least cost of computing the join of these relations. Other, more complex estimates, such as total disk I/O's, could be used if we were willing and able to do the extra calculation involved.*

- *The expression that yields the least cost. This expression joins the set of relations in question, with some grouping. We can optionally restrict ourselves to left-deep expressions, in which case the expression is just an ordering of the relations.*

2.8.3 Greedy algorithm for join order

Even the carefully limited search of dynamic programming leads to a number of calculations that is exponential in the number of relations joined. It is reasonable to use an exhaustive method like dynamic programming or branch-and-bound search to find optimal join orders of five or six relations. However, when the number of joins grows beyond that, or if we choose not to invest the time necessary for an exhaustive search, then we can use a join-order heuristic in our query optimizer.

The most common choice of heuristic is a greedy algorithm, where we make one decision at a time about the order of joins and never backtrack or reconsider decisions once made. We shall consider a greedy algorithm that only selects a left-deep tree. The "greediness" is based on the idea that we want to keep the intermediate relations as small as possible at each level of the tree.

Start with the pair of relations whose estimated join size is smallest. The join of these relations becomes the current tree. Find, among all those relations not yet included in the current tree, the relation that, when joined with the current tree, yields the relation of smallest estimated size. The new current tree has the old current tree as its left argument and the selected relation as its right argument.

2.9 Physical Query Plan

We have parsed the query, converted it to an initial logical query plan, and improved that logical query plan with transformations. Part of the process of selecting the physical query plan is enumeration and cost estimation for all of our options, then

focused on the question of enumeration, cost estimation, and ordering for joins of several relations. By extension, we can use similar techniques to order groups of unions, intersections, or any associative/commutative operation

There are still several steps needed to turn the logical plan into a complete physical query plan.

- *Selection of algorithms to implement the operations of the query plan, when algorithm-selection was not done as part of some earlier step such as selection of a join order by dynamic programming.*
- *Decisions regarding when intermediate results will be materialized (created whole and stored on disk) , and when they will be pipelined (created only in main memory, and not necessarily kept in their entirety at any one time) .*
- *Notation for physical-query-plan operators, which must include details regarding access methods for stored relations and algorithms for implementation of relational-algebra operators.*

2.9.1 Choosing a Selection Method

One of the important steps in choosing a physical query plan is to pick algorithms for each selection operator.

Assuming there are no multidimensional indexes on several of the attributes, then each physical plan uses some number of attributes that each have an index, and are compared to a constant in one of the terms of the selection. We then use these indexes to identify the sets of tuples that satisfy each of the conditions. For simplicity, we shall not consider the use of several indexes in this way. Rather, we limit our discussion to physical plans that:

- *Use one comparison of the form $A\theta c$, where A is an attribute with an index, c is a constant, and θ is a comparison operator such as $=$ or $<$.*

- Retrieve all tuples that satisfy the comparison, using the index scan physical operator.
- Consider each tuple selected to decide whether it satisfies the rest of the selection condition. We shall call the physical operator that performs this step *Filter*; it takes the condition used to select tuples as a parameter, much as the *rr* operator of relational algebra does.

In addition to physical plans of this form, we must also consider the plan that uses no index but reads the entire relation (using the table-scan physical operator) and passes each tuple to the *Filter* operator to check for satisfaction of the selection condition.

We decide among the physical plans with which to implement a given selection by estimating the cost of reading data for each possible option. To compare costs of alternative plans we cannot continue using the simplified cost estimate of intermediate-relation size. The reason is that we are now considering implementations of a single step of the logical query plan, and intermediate relations are independent of implementation. Thus, we shall refocus our attention and resume counting disk I/O's.

2.9.2 Choosing a Join Method

On the assumption that we know (or can estimate) how many buffers are available to perform the join, we can apply the formulas for sort/ indexed/ hash join. However, if we are not sure of, or cannot know, the number of buffers that will be available during the execution of this query (because we do not know what else the DBMS is doing at the same time), or if we do not have estimates of important size parameters such as the $V(R, a)$'s, then there are still some principles we can apply to choosing a join method. Similar ideas apply to other binary operations such as unions, and to the full-relation, unary operators, γ, δ

2.9.3 Pipelining Versus Materialization

The last major issue we shall discuss in connection with choice of a physical query plan is pipelining of results. The naive way to execute a query plan is to order the operations appropriately (so an operation is not performed until the argument(s) below it have been performed), and store the result of each operation on disk until it is needed by another operation. This strategy is called materialization, since each intermediate relation is materialized on disk. A more subtle, and generally more efficient, way to execute a query plan is to interleave the execution of several operations. The tuples produced by one operation are passed directly to the operation that uses it, without ever storing the intermediate tuples on disk. This approach is called pipelining, and it typically is implemented by a network of iterators whose functions call each other at appropriate times. Since it saves disk I/O 's, there is an obvious advantage to pipelining, but there is a corresponding disadvantage. Since several operations must share main memory at any time, there is a chance that algorithms with higher disk-I/O requirements must be chosen, or thrashing will occur, thus giving back all the disk-I/O savings that were gained by pipelining, and possibly more.

2.9.4 Pipelining Unary Operations

*Unary operations - selection and projection - are excellent candidates for pipelining. Since these operations are tuple-at-a-time, we never need to have more than one block for input, and one block for the output. We may implement a pipelined unary operation by iterators, The consumer of the pipelined result calls **GetNext()** each time another tuple is needed. In the case of a projection, it is only necessary to call **GetNext()** once on the source of tuples, project that tuple appropriately, and return the result to the consumer. For a selection σ_C (technically, the physical operator **Filter(C)**), it may be necessary to call **GetNext()** several times at the source,*

until one tuple that satisfies condition c is found.

2.9.5 Pipelining Binary Operations

The results of binary operations can also be pipelined. We use one buffer to pass the result to its consumer, one block at a time. However, the number of other buffers needed to compute the result and to consume the result varies, depending on the size of the result and the sizes of other relations involved in the query. We shall use an extended example to illustrate the tradeoffs and opportunities.

2.9.6 Notation for Physical Query Plans

Operators for Leaves

Physical Operators for Selection

Physical Sort Operators

Other Relational-Algebra Operations

2.9.7 Ordering of Physical Operations

Our final topic regarding physical query plans is the matter of order of operations. The physical query plan is generally represented as a tree, and trees imply something about order of operations, since data must flow up the tree. However, since bushy trees may have interior nodes that are neither ancestors nor descendants of one another, the order of evaluation of interior nodes may not always be clear. Moreover, since iterators can be used to implement operations in a pipelined manner, it is possible that the times of execution for various nodes overlap, and the notion of "ordering" nodes makes no sense.

If materialization is implemented in the obvious store-and-later-retrieve way, and pipelining is implemented by iterators, then we may establish a fixed sequence of events

whereby each operation of a physical query plan is executed. The following rules summarize the ordering of events implicit in a physical query-plan tree:

- Break the tree into subtrees at each edge that represents materialization. The subtrees will be executed one-at-a-time.
- Order the execution of the subtrees in a bottom-up, left-to-right manner. To be precise, perform a preorder traversal of the entire tree. Order the subtrees in the order in which the preorder traversal exits from the subtrees.
- Execute all nodes of each subtree using a network of iterators. Thus, all the nodes in one subtree are executed simultaneously, with *GetNext* calls among their operators determining the exact order of events.

2.10 Introduction to Data Streams

2.11 Data stream windowing

2.12 Query Processing of data streams(Combine the DBMS and DSMS)

2.13 Challenges of query optimization on data streams

2.14 Conclusion and discussion

Chapter 3

Stream Optimization

3.1 Query Optimization of Data Streams

Chapter 4

Implementation

4.1 Mathematics

Chapter 5

Evaluation

5.1 Measures used

Chapter 6

Conclusion and Further work

6.1 Conclusion

Appendix A

Proof of xyz

This is the appendix.

CURRICULUM VITAE

Joe Graduate

Basically, this needs to be worked out by each individual, however the same format, margins, typeface, and type size must be used as in the rest of the dissertation.