ENST Paris - Promotion 2006
Bogdan ALEXE

Rapport de stage d'ingenieur

An Alternative Storage Scheme for the DBNotes Annotation Management System for Relational Databases

Non Confidentiel

Directeur de stage: Wang-Chiew Tan
Correspondant de stage: Talel Abdessalem
Dates de stage: 12 Juillet - 12 Decembre 2005
Adresse de l'organisme:
University of California, Santa Cruz
Jack Baskin School of Engineering
Computer Science Department
1156 High Street
Santa Cruz 95064, California
United States of America

Abstract

DBNotes is an annotation management system for relational databases developed in the Database Research Group at University of California, Santa Cruz. Every column of every tuple in every relation of a database can have one or more annotations, which are pieces of information relevant to the data, but normally not stored with the data in the database. In the initial implementation of DBNotes, annotations are physically stored in the database in a scheme that presents several disadvantages. This report presents work carried on the DBNotes system, dealing with the design and implementation of an alternative storage scheme for annotations that attempts to eliminate the disadvantages of the initial scheme (mainly the data redundancy). It also presents experimental results indicating the absolute performance of this storage scheme and comparisons that prove it is more efficient than the scheme considered in the initial implementation of the system.

DBNotes est un systeme de gestion des annotations pour les bases de donnees relationnelles, developpe dans le Groupe de Recherche dans les Bases de Donnees a l'Universite de Californie a Santa Cruz. Chaque colonne de chaque tuple de chaque relation dans une base de donnees peut avoir une ou plusieurs annotations, qui sont des informations pertinentes pour les donnees, mais qui normalement ne sont pas stockees a cote des donnees dans la base. Dans l'implementation initiale de DBNotes, les annotations sont physiquement presentes dans la base de donnees suivant un schema qui presente plusieurs inconvenients. Ce rapport presente le travail effectue sur le systeme DBNotes, pour la conception et l'implementation d'un schema alternative pour le stockage des annotations qui essaye d'eliminer les inconvenients du schema initial (principalement la redondance). Il presente aussi des resultats experimentaux qui indiquent les performances absolues de ce schema de stockage et des comparaisons qui prouvent qu'il est plus efficace par rapport au schema considere dans l'implementation initiale du systeme.

Acknowledgments

I would like to thank Professor Wang-Chiew Tan for offering me the opportunity to work in the Database Research Group, for all the advice and the good ideas she gave me, and for the continuous guidance all along the period I spent working on DBNotes at UC Santa Cruz.

I also thank Laura Chiticariu for the insight she gave me into the initial implementation of the DBNotes system.

Contents

1 Introduction

In 2005, between July and December, I worked in the Database Research Group at the University of California, Santa Cruz. My primary focus was on the annotation management system for relational databases DBNotes. The goal of this research project is to provide a means for annotations (pieces of information about the actual data, that normally would not be stored in the database) to be efficiently stored, retrieved and propagated as the data is transformed by queries. This kind of system that allows the easy manipulation of annotation has many applications: storing comments about the data, error reports, accuracy levels in scientific applications, tracing the provenance of data.

The way annotations are propagated is established through a PROPAGATE clause that is added to queries to form a variant of SQL called pSQL. The DBNotes system is built on top of a relational database management system. It translates an incoming pSQL query into a union of SQL queries that are sent to the database engine. The provided output is postprocessed and the final answer is given to the user.

My main contributions reside in the design and implementation of an alternative storage scheme for the annotations in DBNotes whose aim is to compensate the disadvantages of the initial scheme, as well as in the experimental evaluation of its performance.

This report is organized as follows. First, some information about the University of California at Santa Cruz in general and the Database Research Group in particular is given. The DBNotes system is introduced, together with its motivation. Then, the pSQL language that allows the manipulation of annotations and their propagation under various schemes is described. The architecture of the system is given next. The following section deals with the alternative annotation storage scheme that was implemented: the relational schema, the translation algorithm from custom pSQL to SQL and the experimental evaluation (absolute performance and a comparison to the initial storage scheme). Finally, some remarks are given about a data exchange project I am involved in. Conclusions are presented at the end of this report.

2 The Environment of My Work

2.1 Overview of UC Santa Cruz

The University of California, Santa Cruz, opened in 1965 and grew to its current enrollment of about 15,000 students. Undergraduates pursue 61 majors in physical and biological sciences, social sciences, humanities and arts. Graduate students work toward graduate certificates, master's degrees, or Ph.D. degrees in 32 academic fields.

Faculty members at UC Santa Cruz include 10 members of the National Academy of Sciences, 19 members of the American Academy of Arts and Sciences, and two members of the Institute of Medicine.

All undergraduates, whether they live on campus or off, are affiliated with one of the UCSC colleges (Cowell, Stevenson, Crown, Merrill, Porter, Kresge, Oakes, Eight, Nine, and Ten). Although students take classes in any number of colleges and academic units throughout the campus, core courses within each college provide a common academic base for first-year and transfer students. At the conclusion of work in their major, all UCSC seniors must pass a comprehensive examination or, in some majors, complete a senior thesis or equivalent body of work.

Graduate study began at UCSC in 1966 with programs in astronomy, biology, and history of consciousness, a program that combines the humanistic disciplines with links to the social sciences, natural sciences, and arts. In 1967, graduate programs in chemistry, literature, and Earth sciences were introduced. Additional graduate programs have been established in anthropology, computer engineering, computer science, economics, education, environmental studies, history, linguistics, mathematics, music, physics, psychology, science communication, sociology, and theater arts. In 1998, UCSC began a Ph.D. program in ocean sciences. In 1999, M.S. and Ph.D. programs in environmental toxicology and a Ph.D. program in politics began recruiting students. In 2000, M.A. and Ph.D. programs in philosophy were launched. In 2002, a new Ph.D. program in education began recruiting students. In 2003, a new M.A. program in history and an M.F.A. program in digital arts and new media began. In 2004, additional programs in social documentation and musical composition were established.

In 1997, UCSC began its first professional school, the Jack Baskin School of Engineering, and introduced a new undergraduate electrical engineering major, followed in 1998 by a major in information systems management. In 1997-98, UCSC began offering a "distance-learning" version of

the M.S. in computer engineering, with a concentration in networking engineering, at its Silicon Valley facilities. In 2001, an undergraduate major in bioinformatics was launched and, in 2003, M.S. and Ph.D programs in bioinformatics were initiated. The school is in the process of hiring faculty, which has led to the new Electrical Engineering Department; an Applied Mathematics and Statistics Department is also being established. In 2003, retired engineer and philanthropist Jack Baskin gave additional funding for a new engineering building, Engineering 2, and to create an endowed chair in the Department of Biomolecular Engineering, which began in 2004.

In conjunction with graduate teaching and intellectual inquiry, the campus is home to two Organized Research Units: the Institute of Marine Sciences and Santa Cruz Institute for Particle Physics. The University of California Observatories/Lick Observatory is a Multicampus Research Unit headquartered at UCSC. UC's Institute of Geophysics and Planetary Physics (IGPP), a Multicampus Research Unit, includes a branch on the UCSC campus established in 1999. UCSC also is one of several UC campuses sponsoring the Institute for Quantitative Biomedical Research (QB3), and the Center for Information Technology Research in the Interest of Society (CITRIS), two of the California Institutes for Science and Innovation established in 2000. Over the years, UCSC has been awarded a total of $863 million for contracts and grants within these units and in numerous other campus research programs.

The ten UCSC colleges–each a separate community with its own buildings and administration– are built around a core of shared university facilities. These include the main and science/engineering libraries, performing arts buildings, visual arts studios, classrooms, computer facilities, and a complex of highly specialized buildings for the physical & biological sciences, and engineering. Athletic facilities exist on the east and west sides of the campus.

Significant private funds–over $242 million through the 2003-04 year– have been donated to build or enhance academic, student-life, and other facilities at the campus, as well to fund programs, research, and scholarships.

UCSC is increasing both its enrollment and resources and diversifying its educational and research opportunities over the next decade. In pace with this growth will be the development of additional academic programs. Another major objective is to provide for a growing number of students, faculty, and staff of diverse ethnic backgrounds and cultural experiences.

UCSC is moving forward with planning its Silicon Valley Center in order to respond to UC's

increasing enrollment; intensify partnerships with the area's community colleges, state universities, and businesses; and develop distance learning. A 10-year, $30 million program establishing a University Affiliated Research Center (UARC) at NASA Ames was initiated in 2003. UARC provides research and educational capabilities to meet NASA's requirements and to develop future human resources in science and technology.

2.2 The Database Research Group

My work was carried under the supervision of Prof. Wang-Chiew Tan, in the Database Research Group, part of the Computer Science Department in the Jack Baskin School of Engineering.

This research group consists of three faculty members and five graduate students. The main research topics revolve around database theory, data provenance, annotations, data exchange, information integration, query processing for semi-structured data and advanced visual query interfaces.

3 DBNotes

3.1 Introduction and Motivation

In many applications, there is some information normally not stored in a database, but associated with the data in the database. In many cases, this additional information must be propagated along with the data it is associated with through query transformations. One example is the situation where scientific data is associated with metadata describing its accuracy or reliability. Another one is a situation where one might be interested in analyzing the chain of transformations a piece of data has gone through before arriving in an output relation.

DBNotes is an annotation management system for relational databases. Every "cell" in the database (every column of every tuple) can have zero or more annotations. An annotation can represent information about data provenance, comments or some other kind of metadata. As queries transform the data, annotations must be propagated from the source instance to the result instance of a query application. By default, annotations are propagated based on where the data is coming from. Thus tracing the provenance of data is one direct application (every cell is annotated in this case with its "location" in the database). A second use of annotations is to store any additional information about the data that otherwise would not be stored inside the database. Examples include comments, error reports, security level or quality of data.

In DBNotes, there are three possible schemes for propagating annotations: default, default-all and custom. Under the default scheme, annotations are propagated in the result of a query based on where they originated from (if the query copies data from a source location to a target location, then the annotations for the source location are propagated to the target location). Unfortunately, for some applications, this propagation scheme is not sufficient. The result of propagating annotations under the default scheme can be different based on how the query is written. The default-all scheme deals with this by propagating annotations based on all possible equivalent formulations of a query. Under this scheme, the propagation result will always be the same for equivalent queries. The third propagation scheme (custom) allows the user to completely specify the way annotations are propagated.

3.2 pSQL and Propagation Schemes

The queries considered for transforming the data belong to the Select-Project-Join-Union fragment from SQL (this fragments corresponds to conjunctive queries with union). The SQL syntax is

extended with a PROPAGATE clause that specifies the way annotations are propagated.

A pSQL query is an expression of the form Q_1 UNION Q_2 UNION $\ldots Q_n$ where each Q_i is a query of the following form:

SELECT DISTINCT	*selectlist*
FROM	*fromlist*
WHERE	*wherelist*
PROPAGATE	DEFAULT | DEFAULT_ALL |
	$r_1.A_1$ TO B_1, ..., $r_n.A_n$ TO B_n

The *fromlist* has the form $R_1\ r_1, \ldots, R_k\ r_k$. In this expression, r_i is a tuple variable in the relation R_i. The *selectlist* has the form $r_1.C_1$ AS D_1, ..., $r_m.C_m$ AS D_m. In this expression, r_i are variables defined in *fromlist*, C_i is an attribute in R_i and D_i is a name for the corresponding attribute in the relation representing the output of this query. If the WHERE clause is present, its *wherelist* represents a conjunction of equalities between relation attributes or between constants and relation attributes.

The PROPAGATE clause specifies the way annotations are propagated. Its parameter can be either DEFAULT, DEFAULT_ALL or a list of expressions of the form $r.A$ TO B where A is the name of an attribute in the tuple bound to the variable r and B is a name among the output attributes D_i.

The semantics of a pSQL query are similar to a normal SQL query, with the addition of propagating the annotations according to the PROPAGATE clause.

In the following, a *location* will be a triple (r, t, i) representing the attribute at position i in tuple t of relation r. If the relation r is clear from the context, we can also use the simplified notation (t, i). The notation $\mathcal{A}(r, t, i)$ will represent the set of annotation at the location (r, t, i). Similarly, for a location (t, i), $\mathcal{A}(t, i)$ represents the set of annotations at this location.

Under the custom propagation scheme, the user has the possibility to completely specify the way annotations are propagated. Let's consider a query where the propagate clause has the form $r_1.A_1$ TO B_1, ..., $r_n.A_n$ TO B_n. The semantics of this query is the following: for every binding of the tuple variables according to the *fromlist* such that *wherelist* is satisfied, a tuple t is emitted according to *selectlist*. Every clause of the form $r_i.A_i$ TO B_i in the PROPAGATE clause determines the annotations at the source location (r_i, A_i) to be propagated to the output location (t, B_i). Any

duplicate tuples in the output relation are merged. In the output relation, the set of annotations at a given location is $\mathcal{A}(s, B) = \cup_{t_j=s}\mathcal{A}(t_j, B)$, where t_j are duplicate tuples that are merged in the tuple s.

In the following, the values in curly brackets along with the data in a cell represent the annotations for that cell.

As an example, let's consider the relation:

R	ID	Desc
	g231 $\{a_{11}\}$	AB $\{a_{12}\}$
	g756 $\{a_{13}\}$	CC $\{a_{14}\}$

and the query:

SELECT DISTINCT	r.ID as ID, r.Desc as Desc
FROM	R r
PROPAGATE	r.ID to ID, r.Desc to ID

The result of evaluating the query with custom propagation on the relation is the following:

	ID	Desc
	g231 $\{a_{11}, a_{12}\}$	AB
	g756 $\{a_{13}, a_{14}\}$	CC

Under the default propagation scheme, the annotations at a given location in the output relation originate at the locations in the source relation where the data is copied from. We can define the semantics of a pSQL query with default propagation in the following way: for every binding of the tuple variables according to the *fromlist* such that *wherelist* is satisfied, a tuple t is emitted according to *selectlist*, together with the sets of annotations for every location in t. As in the previous propagation scheme, duplicate tuples are merged and the set of annotations for a location in a merged tuple s is the union of the sets of annotations for that location in every tuple t_j merged into s.

For the default propagation scheme, we can consider the following example relation:

R	ID	Desc
	z131 $\{a_1\}$	AB $\{a_2\}$
	q229 $\{a_3\}$	CC $\{a_4\}$
	q939 $\{a_5\}$	ED $\{a_6\}$

and the query:

SELECT DISTINCT	r.ID as ID, r.Desc as Desc
FROM	R r
WHERE	r.ID = "q229"
PROPAGATE	DEFAULT

The result of evaluating the query with default propagation on the relation is the following:

	ID	Desc
	g229 $\{a_3\}$	CC $\{a_4\}$

Two pSQL queries are equivalent if they produce the same output on all databases. Two pSQL queries are annotation-equivalent if they produce the same annotated output on all databases. There are situations where two pSQL queries are equivalent, but not annotation-equivalent. This motivates the definition of a third propagation scheme: default-all. Under this propagation scheme, the annotated output of a pSQL query is invariant with respect to the way the query is formulated. The annotations are propagated according to the default propagation scheme for all the equivalent formulations of the query. The resulting tuples are merged in the end. Although there might be an infinite number of equivalent formulations for a query, there is a method of computing the output by examining only a finite set of queries (the *query basis* of the original query).

We consider the following example relation for the default-all propagation scheme:

R	ID	Name
	p332 $\{a_7\}$	AB $\{a_8\}$
	p916 $\{a_9\}$	AB $\{a_{10}\}$

and the query:

SELECT DISTINCT	r.ID as ID, r.Name as Name
FROM	R r
PROPAGATE	DEFAULT ALL

The result of evaluating the query with default-all propagation on the relation is the following:

	ID	Desc
	p332 $\{a_7\}$	AB $\{a_8, a_{10}\}$
	p916 $\{a_9\}$	AB $\{a_8, a_{10}\}$

The result above is justified by the fact that the following query is equivalent to the query above (without the propagation clause):

SELECT DISTINCT	r.ID as ID, r.Name as Name
FROM	R r, R q
WHERE	r.Name = q.Name

Under the default-all propagation scheme, annotations for identical values in the Name column are combined.

3.3 The DBNotes System and the Initial Storage Scheme

The DBNotes system consists of two main modules: the translator and the postprocessor. The input of the translator module is a pSQL query and its output is a union of SQL queries to be sent to a relational database management system. These queries are executed by the database system and the sorted result is given as input to the postprocessor module. This module, in a single pass through the sorted tuples, merges identical tuples and performs the union of the sets of annotations on these tuples.

The initial scheme for storing annotations in DBNotes was the following: for every attribute A in a relation R, there is an extra attribute A_a used to store annotations for A. The modified relation is denoted by R'. For instance, if the initial relation is $R(A, B)$, the relation modified to store annotations is $R'(A, A_a, B, B_a)$. If a tuple t contains k annotations $\{a_1, \ldots, a_k\}$ for a given location (t, A), then the annotated relation will contain k tuples $t_1, \ldots, t_k$ such that $t_i.A_a = a_i$. If we consider the tuple (a $\{a_1, a_2\}$, b $\{b_1\}$) (the values in the curly braces are the annotations), the following tables are valid relational representations of it:

A	A_a	B	B_a
a	a_1	b	b_1
a	a_2	b	-

A	A_a	B	B_a
a	-	b	b_1
a	a_1	b	-
a	a_2	b	-

We will consider an example in order to illustrate the function of the postprocessor. If the relation returned by the database engine is the following:

	A	A_a	B	B_a
	a	a_1	b	a_2
	a	a_3	b	-
	a	-	c	a_2

then the postprocessor will merge the tuples with identical data values and return the following tuples as a result: $(a\ \{a_1, a_3\}, b\ \{a_2\})$, $(a\ \{\}, c\ \{a_2\})$.

4 Alternative Storage Scheme for DBNotes

4.1 Schema Description

In the initial annotation storage scheme, if a location in a tuple had multiple annotations, the annotated relation contained multiple copies of that tuple. This leads to redundant storage of some of the tuples in the database. A direct consequence of this is an increase in the response time for queries involving annotated relations (especially queries with joins).

The goal here is to design a new annotation storage scheme that would increase the efficiency of running queries over annotated relations by eliminating the redundancies in the data relations. In what follows, we will describe this alternative storage scheme and will provide some experimental results that provide insight into the performance of this new scheme and compare the behavior of the initial and alternative storage schemes.

The alternative storage scheme uses the concept of *rowid*, present in many available database management systems, such as Oracle or Postgres. The *rowid* is a value that uniquely identifies a tuple in an entire database.

In the alternative storage scheme, for every data relation R in the database, there will exist one annotation relation R_a that will store the annotations for the locations in R. Let's consider the case where the database contains the relation R with attributes $(A_1, A_2, \ldots, A_n)$. The relation R_a containing the annotations for R will have the attributes $(\text{rowid}, A_{1a}, A_{2a}, \ldots, A_{na})$. The column rowid in R_a will contain rowids for tuples in R, and the columns A_{ia} will contain annotations for locations in columns A_i of R.

As an example, let's consider the situation where we have a single annotated tuple

$$t = (a\ \{a_1, a_2\}, b\ \{b_1\})$$

The first element in the tuple corresponds to attribute A and the second to attribute B. The relations R and R_a for this example have the following form:

R	A	B
t	a	b

R_a	rowid	A_a	B_a
	rowid(t)	a_1	b_1
	rowid(t)	a_2	-

4.2 Translation Algorithm

The queries are ultimately run by a relational database system which accepts SQL as its query language. Therefore, the pSQL queries have to be translated to equivalent unions of SQL queries before being sent to the database system. Because pSQL queries with default and default-all propagation clauses can be translated to queries with custom propagation, we only need one translation mechanism: from pSQL queries with custom propagation clauses to equivalent unions of SQL queries.

The translation from custom pSQL consists of two phases. First, a set of SQL queries is generated. Second, a wrapper query is created that takes the union of the queries generated in the first phase and sorts the output of these queries. This step is required by the postprocessor module (otherwise, it would need multiple passes through the tuples).

As we have seen before, a pSQL query with custom propagation has the following form:

SELECT DISTINCT	$r_1.C_1$ AS D_1, ..., $r_m.C_m$ AS D_m
FROM	$R_1\ r_1, \ldots, R_k\ r_k$
WHERE	*wherelist*
PROPAGATE	$r_1.A_1$ TO B_1, ..., $r_n.A_n$ TO B_n

This query has to be translated into a union of SQL queries. To comply with the input format of the postprocessor, the output relation of each of these SQL queries has to contain two columns for each of the D_i output attributes: one for the data and one for the annotations that might be propagated by the query to D_i. The B_i annotation destinations in the propagation clause are attributes among the selected attributes D_i. First we need to establish on which output attribute will arrive each of the propagated annotations. So we need to set up a correspondence between each of the output attributes (D_i) and the list of locations among the $r_j.A_j$ that will contribute with annotations to D_i. This will give us the annotation sources for each annotation destination D_i. We can represent the correspondence as follows:

$$\begin{array}{ccl} D_1 & \longleftarrow & A_{11}, A_{12}, \ldots, A_{1p_1} \\ D_2 & \longleftarrow & A_{21}, A_{22}, \ldots, A_{2p_2} \\ \vdots & & \\ D_m & \longleftarrow & A_{m1}, A_{m2}, \ldots, A_{mp_m} \end{array}$$

We take one annotation source (A_{ij}) from each list, generate a SQL query as we will see below and repeat until all the annotation sources have been considered.

As we have mentioned before, the output relation of each SQL query will contain two columns for each output attribute of the pSQL query. One column will contain the data and appears the same in each of the generated SQL queries. The second column contains the annotations for that output attribute. In the SELECT clause of the SQL query, this column can be either the attribute where the annotations are extracted from, or a NULL (if there are no annotations to be propagated to this specific output attribute, or all the sources for this attribute have already been considered in previous SQL queries).

The FROM clause of each SQL query contains the list of relations that provide data for the output relation, together with the list of annotation relations that correspond to the annotation sources considered in the SQL query. Each annotation relation is joined to its corresponding data relation based on the equality between the ROW_ID attribute (in the annotation relation) and the ROWID property of each tuple in the data relation. The join to be performed is a LEFT OUTER JOIN, since in the output relation we still need to obtain tuples with no annotations on any of their attributes.

As an example, we can consider the following pSQL query with custom propagation:

```
SELECT DISTINCT   R.Att1 as A, R.Att2 as B, R.Att3 as C
FROM              Test R
PROPAGATE         R.Att1 TO B, R.Att2 TO B, R.Att3 TO C
```

The first part of the translation process produces the following two queries:

Q_1	SELECT DISTINCT	R.Att1 AS A, R.Att2 AS B, R.Att3 AS C, NULL AS C_A, R_ANNOT.Att1_ANNOT AS C_B, R_ANNOT.Att3_ANNOT AS C_C
	FROM	Test R LEFT OUTER JOIN Test_ANNOT R_ANNOT ON R.ROWID=R_ANNOT.ROW_ID
Q_2	SELECT DISTINCT	R.Att1 AS A, R.Att2 AS B, R.Att3 AS C, NULL AS C_A, NULL AS C_C, R_ANNOT.Att2_ANNOT AS C_B
	FROM	Test R LEFT OUTER JOIN Test_ANNOT R_ANNOT ON R.ROWID=R_ANNOT.ROW_ID

Annotations from the Att1 and Att2 attributes are propagated to the B attribute in the output relation (the annotation sources for B are Att1 and Att2) and from the Att3 attribute to the C

output attribute. The query Q_1 propagates the annotations from Att1 to B and from Att3 to C. The query Q_2 propagates to the output attribute B the annotations from its other annotation source (Att2).

Finally, the query to be executed by the database system is the following:

SELECT DISTINCT	*
FROM	(Q_1 UNION Q_2)
ORDER BY	A, B, C

It performs the union of the results given by queries Q_1 and Q_2 and sorts the result so that the postprocessor can merge the tuples with identical data values in a single pass.

4.3 Experimental Evaluation

4.3.1 Experimental Setting

For experimental evaluation, we used the TPC-H benchmark dataset. The schema of this dataset contains eight relations with the following structure:

Part(partkey, name, mfgr, brand, type, size, container, retailprice, comment)
Supplier(suppkey, name, address, nationkey, phone, acctbal, comment)
PartSupp(partkey, suppkey, availqty, supplycost, comment)
Customer(custkey, name, address, nationkey, phone, acctbal, mktsegment, comment)
Orders(orderkey, custkey, orderstatus, totalprice, orderdate, orderpriority,
clerk, shippriority, comment)
Lineitem(orderkey, partkey, suppkey, linenumber, quantity, extendedprice,
discount, tax, returnflag, linestatus, shipdate, commitdate,
receiptdate, shipinstruct, shipmode, comment)
Nation(nationkey,name,regionkey,comment)
Region(regionkey,name,comment)

The queries that were run on this dataset contain an increasing number of joins between tables and an increasing number of output attributes. The queries are denoted as $Q_i(j)$, where i is the number of joins and j is the number of output attributes.

Each of these queries were considered both without a PROPAGATE clause (plain SQL) and with annotation propagation, under the default and default-all schemes. The plain queries are shown below:

$Q_0(1)$	SELECT DISTINCT	s.name
	FROM	Supplier s
$Q_1(1)$	SELECT DISTINCT	s.name
	FROM	Supplier s, Nation n
	WHERE	s.nationkey = n.nationkey
$Q_2(1)$	SELECT DISTINCT	s.name
	FROM	Supplier s, Nation n, Region r
	WHERE	s.nationkey = n.nationkey AND n.regionkey = r.regionkey
$Q_3(1)$	SELECT DISTINCT	s.name
	FROM	Supplier s, Nation n, PartSupp ps, Region r
	WHERE	s.nationkey = n.nationkey AND
		r.regionkey = n.regionkey AND s.suppkey = ps.suppkey
$Q_4(1)$	SELECT DISTINCT	s.name
	FROM	Supplier s, Nation n, PartSupp ps, Region r ,Customer c
	WHERE	s.nationkey = n.nationkey AND r.regionkey = n.regionkey AND
		s.suppkey = ps.suppkey AND s.nationkey = c.nationkey
$Q_0(3)$	SELECT DISTINCT	s.name, s.address, s.phone
	FROM	Supplier s
$Q_1(3)$	SELECT DISTINCT	s.name, s.address, s.phone
	FROM	Supplier s, Nation n
	WHERE	s.nationkey = n.nationkey
$Q_2(3)$	SELECT DISTINCT	s.name, s.address, s.phone
	FROM	Supplier s, Nation n, Region r
	WHERE	s.nationkey = n.nationkey AND r.regionkey = n.regionkey
$Q_3(3)$	SELECT DISTINCT	s.name, s.address, s.phone
	FROM	Supplier s, Nation n, PartSupp ps, Region r
	WHERE	s.nationkey = n.nationkey AND s.suppkey = ps.suppkey AND
		n.regionkey = r.regionkey
$Q_4(3)$	SELECT DISTINCT	s.name, s.address, s.phone
	FROM	Supplier s, Nation n, PartSupp ps, Region r, Customer c
	WHERE	s.nationkey = n.nationkey AND s.suppkey = ps.suppkey AND
		n.regionkey = r.regionkey AND c.nationkey = s.nationkey
$Q_0(5)$	SELECT DISTINCT	s.name, s.address, s.phone, s.acctbal, s.comment
	FROM	Supplier s

$Q_1(5)$	SELECT DISTINCT	s.name, s.address, s.phone,s.acctbal, s.comment
	FROM	Supplier s, Nation n
	WHERE	s.nationkey = n.nationkey
$Q_2(5)$	SELECT DISTINCT	s.name, s.address, s.phone,s.acctbal, s.comment
	FROM	Supplier s, Nation n, Region r
	WHERE	s.nationkey = n.nationkey AND r.regionkey = n.regionkey
$Q_3(5)$	SELECT DISTINCT	s.name, s.address, s.phone,s.acctbal, s.comment
	FROM	Supplier s, Nation n, PartSupp ps, Region r
	WHERE	s.nationkey = n.nationkey AND s.suppkey = ps.suppkey AND
		n.regionkey = r.regionkey
$Q_4(5)$	SELECT DISTINCT	s.name, s.address, s.phone, s.acctbal, s.comment
	FROM	Supplier s, Nation n, PartSupp ps, Region r, Customer c
	WHERE	s.nationkey = n.nationkey AND s.suppkey = ps.suppkey AND
		n.regionkey = r.regionkey AND c.nationkey = s.nationkey

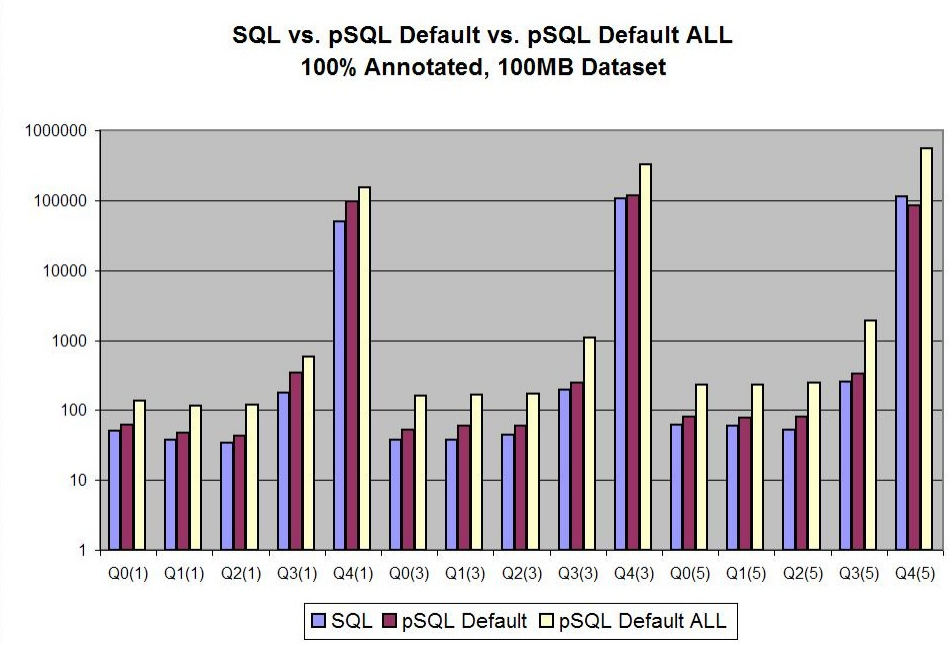
These queries were run on the annotated dataset with different annotation factors: 0%, 30%, 60%, 100%, 130%, 160% and 200%. An annotation factor of 30% means that 30% of the locations in each relation have one annotation and the rest are not annotated, 100% means that each location in each relation has one annotation and 160% means that each location in each relation has one annotation and 60% of the locations have one additional annotation.Two sizes were used for the dataset: 100MB and 500MB.

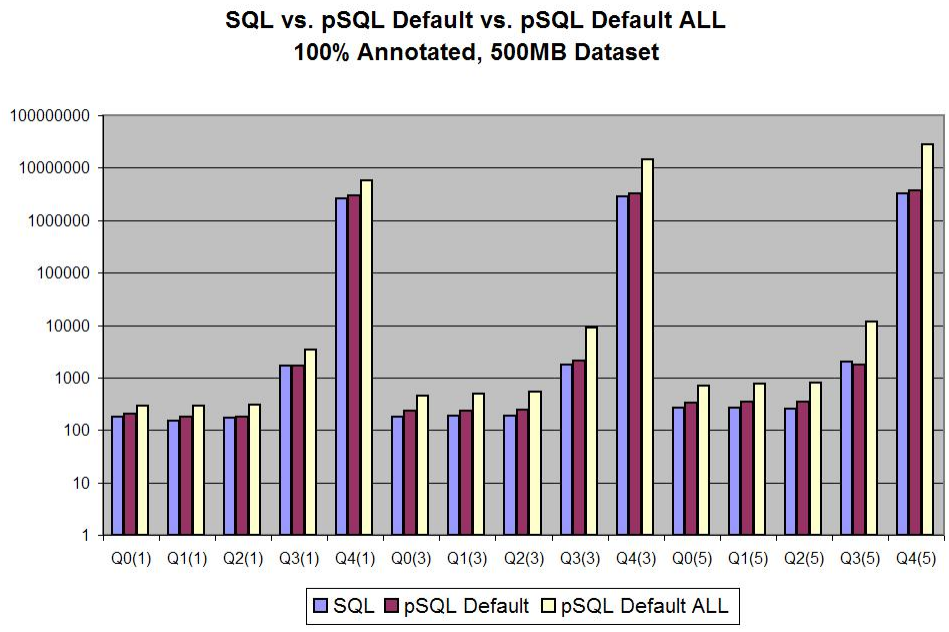
The experiments we conducted had two goals. The first was to determine the behavior of the alternative storage scheme when parameters like the propagation scheme or the number of annotations are changed. The second was to compare the efficiency of the alternative and the initial storage schemas. We expect to obtain a degradation in performance when the number of annotations is increased and also when the default-all propagation scheme is used instead of the default scheme. Our expectation is that the time gain obtained by eliminating the redundancies in the data relations is more important than the loss due to the extra join needed to obtain the annotations, so we expect to prove that the alternative scheme is more efficient than the initial scheme.

4.3.2 Absolute Performance

First, we want to examine the behavior of the system when running pSQL queries with different propagation schemes: no propagation, default and default-all. All the listed queries were run on

the 100% annotated datasets in their plain SQL version and with default and default-all propagate clauses.



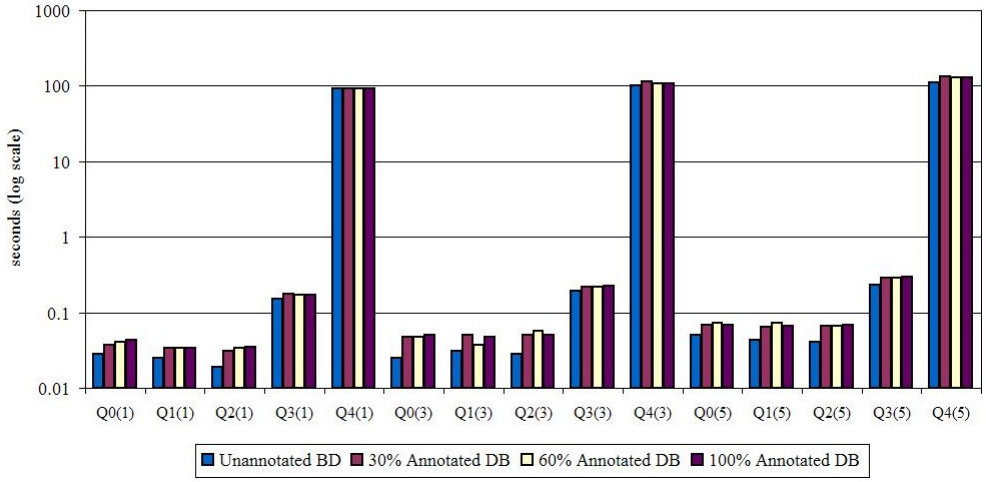


For both the 100MB and the 500MB datasets we observe the same pattern. The only queries that take a considerable amount of time to exacute are the three most expensive ones: $Q_4(1), Q_4(3), Q_4(5)$

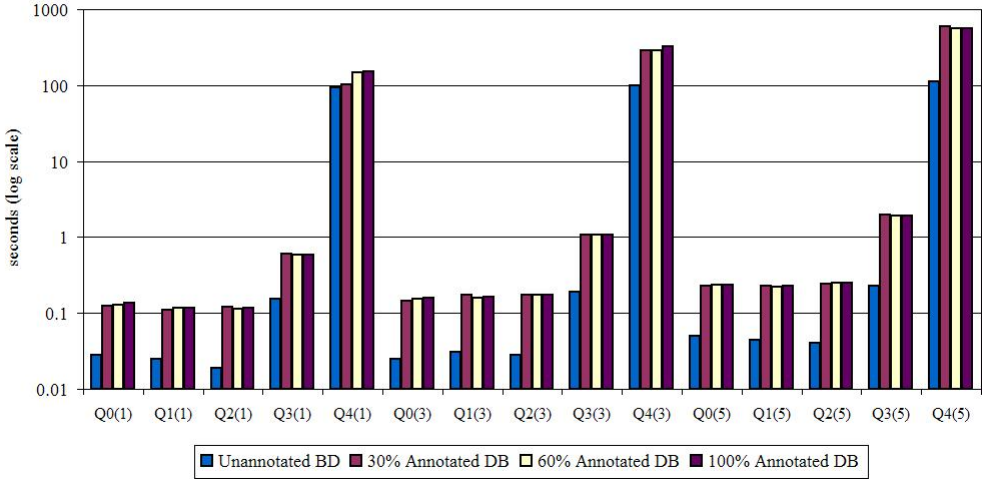
(the queries that contain four joins). Generally, the overhead generated by the default propagation scheme is not very big, as obtaining annotations involves one extra join with the annotation table that has, in our setting, an index on the rowid join attribute (there is an increase by a factor between 1 and 2 in execution times for queries with default propagation compared to plain SQL queries). The gap between the execution times for the default and default-all propagation schemes is more significant, due to the fact that one default-all query is translated into a union of several plain SQL queries to be executed by the database engine.

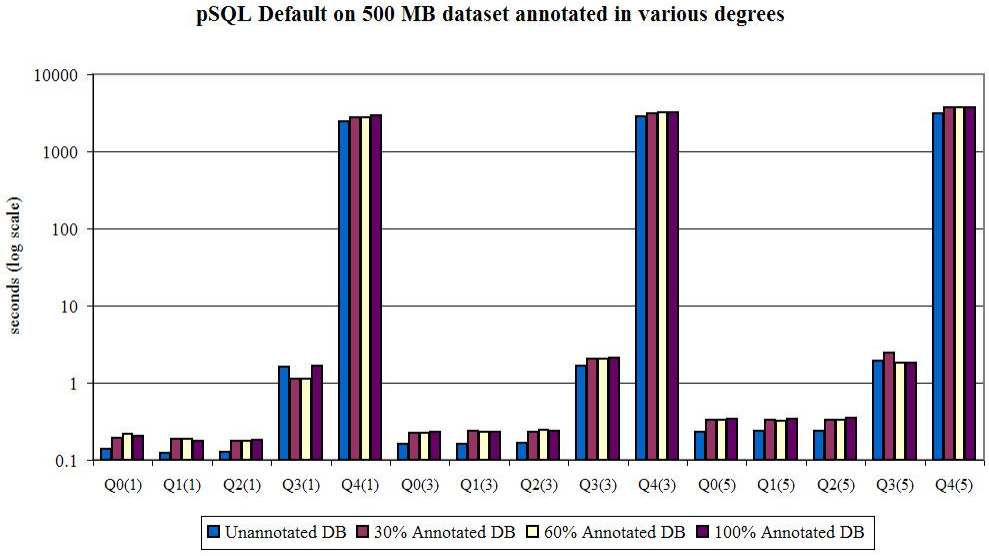
The next step is to run the queries under default and default-all propagation schemes and observe the influence of the number of annotations (here unannotated and 30%,60% and 100% annotated relations).

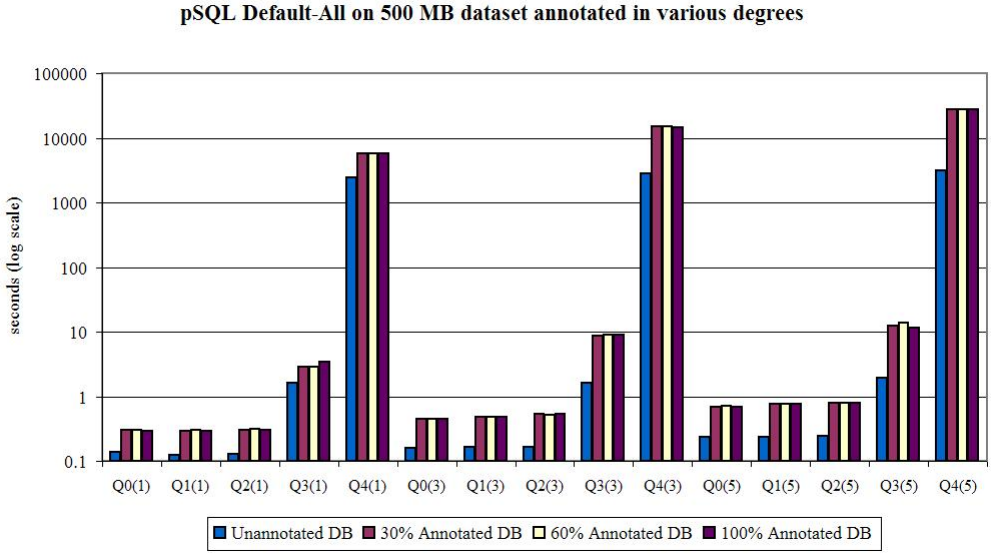
pSQL Default on 100 MB dataset annotated in various degrees



pSQL Default-All on 100 Mb dataset annotated in various degrees

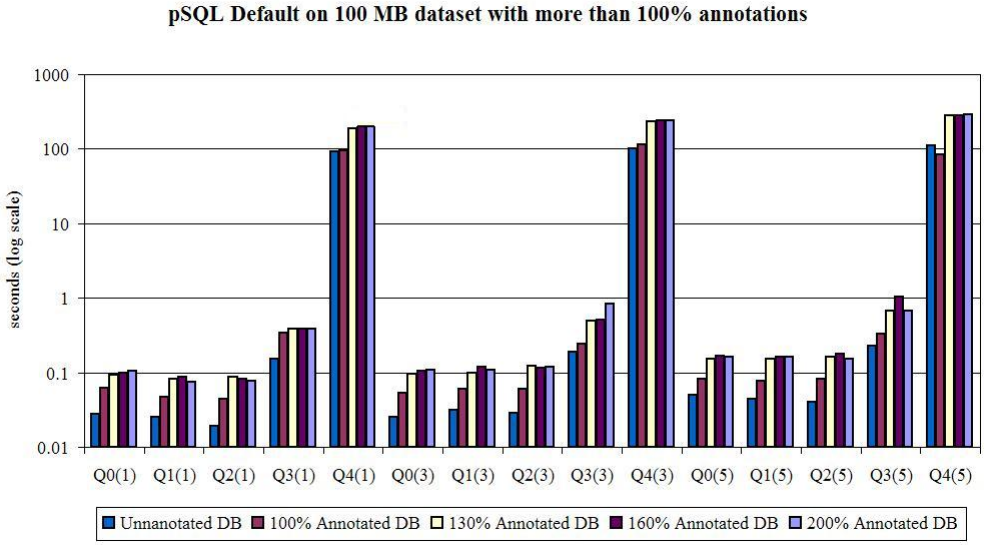


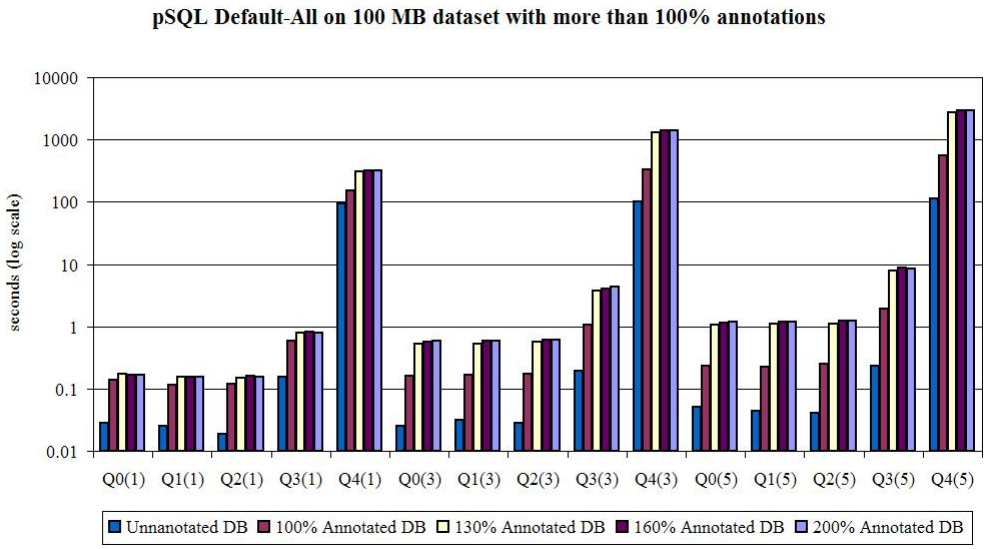




For annotation factors above 0% and below 100% (so for situations where the relations are annotated, but each location has at most one annotation), the number of annotations has little impact on the execution times. For both the 100MB and the 500MB datasets, changing the annotations factor from 30% to 60% and to 100% does not change considerably the execution times for default and default-all queries. For each tuple in the data relation, the annotation relation will contain at most one tuple, for all the situations mentioned here (30%, 60% and 100% annotation factors). Obtaining the annotations for these situations involves joining the data relations with annotation relations of about the same size.

In the next experiment we analyze the behavior of pSQL queries (under default and default-all propagation schemes) on relations annotated above 100% (up until 200%, where each location has two annotations). We used the 100MB dataset.





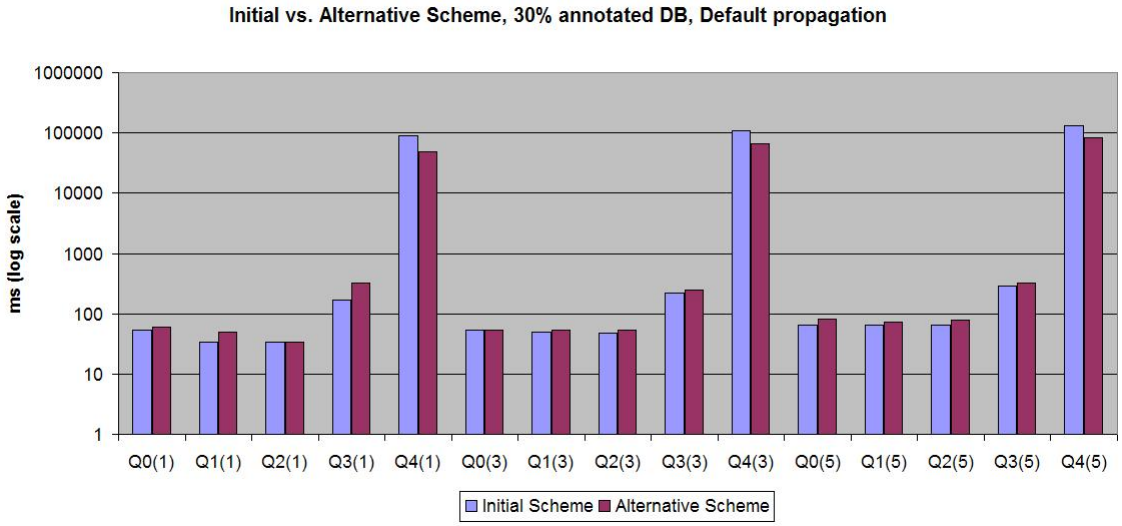
The difference in execution times between queries on relations annotated at 100% and above 100% is more significant. This is due to the fact that for annotation factors above 100%, for each tuple in the data relations there is at least one tuple and at most two tuples in the annotation relations. Because of this, the joins that are necessary for retrieving annotations are more expensive.

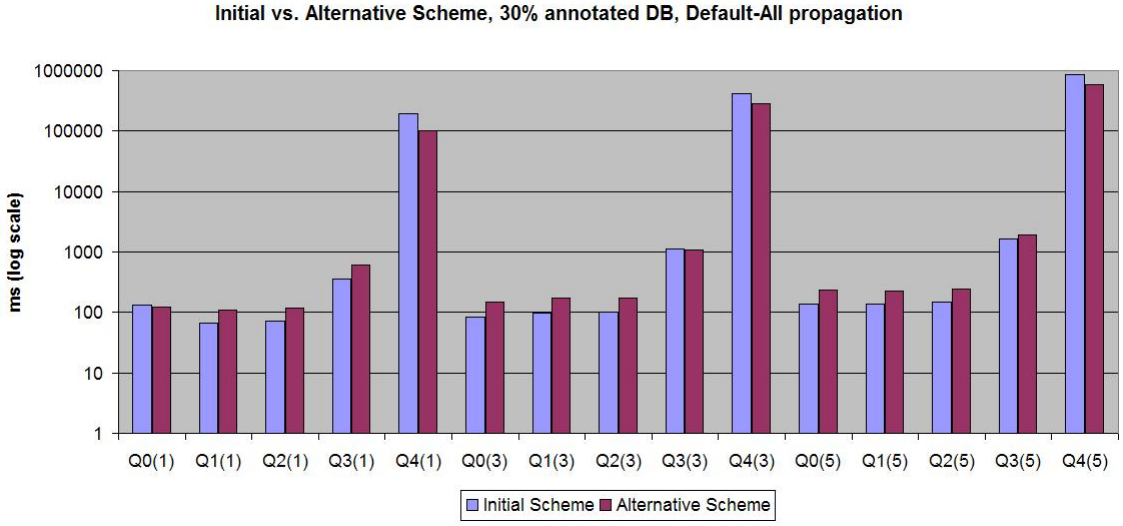
On the other hand, the differences in performance are not significant between 130%, 160% and 200%. This is a consequence of the fact that for all these annotation factors, there is at least one tuple and at most two tuples in the annotation relations for one tuple in the data relation. Thus the cost of joining the data relations with the annotation relations does not change much.

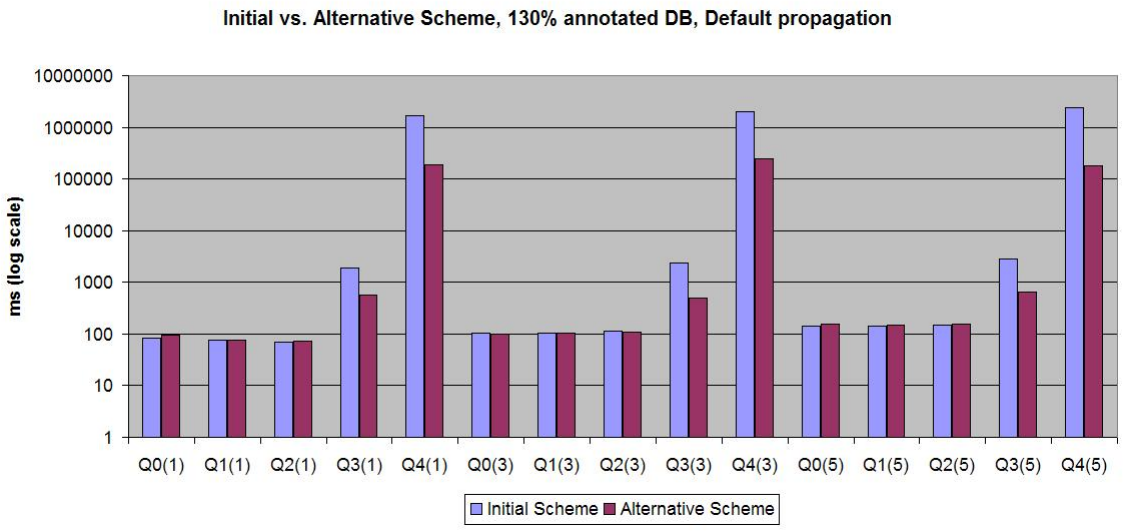
In the graphs above, there are a few situations where the execution times for a query with default propagation scheme is smaller than the time for the same query, without the propagation clause. This counterintuitive observation is justified by the different plans chosen by the database engine for the execution of these queries (a much more inefficient plan for the plain SQL query compared to the one with a default propagation clause).

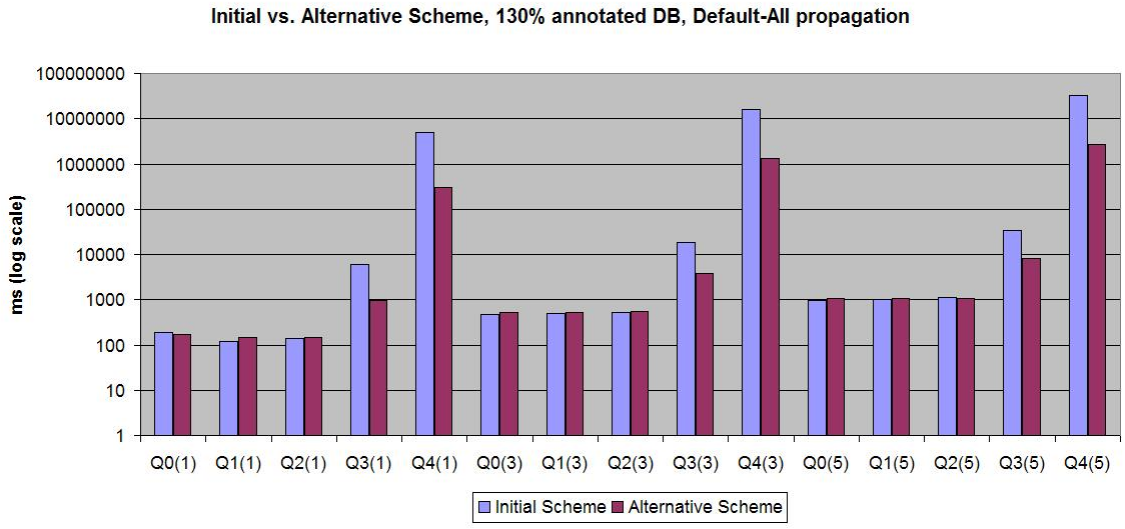
4.3.3 Comparative Results

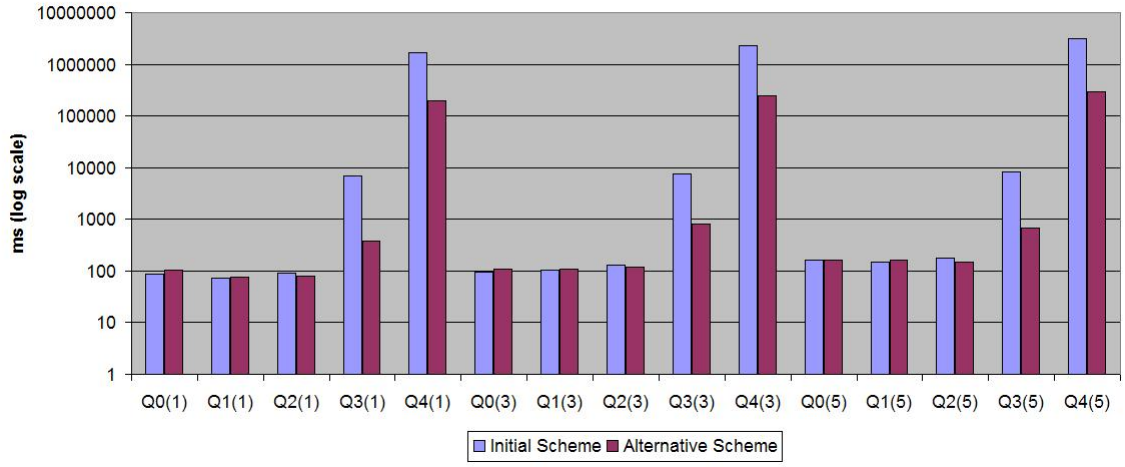
The goal here is to compare the behavior of the initial and the alternative storage scheme. We have to observe how the execution times change when the propagation scheme or the annotation degree change, for both storage schemes. The following graphs present the experimental results for the default and default-all propagation schemes, on the 100MB dataset with increasing values of the annotation degree (30%, 130% and 200%).

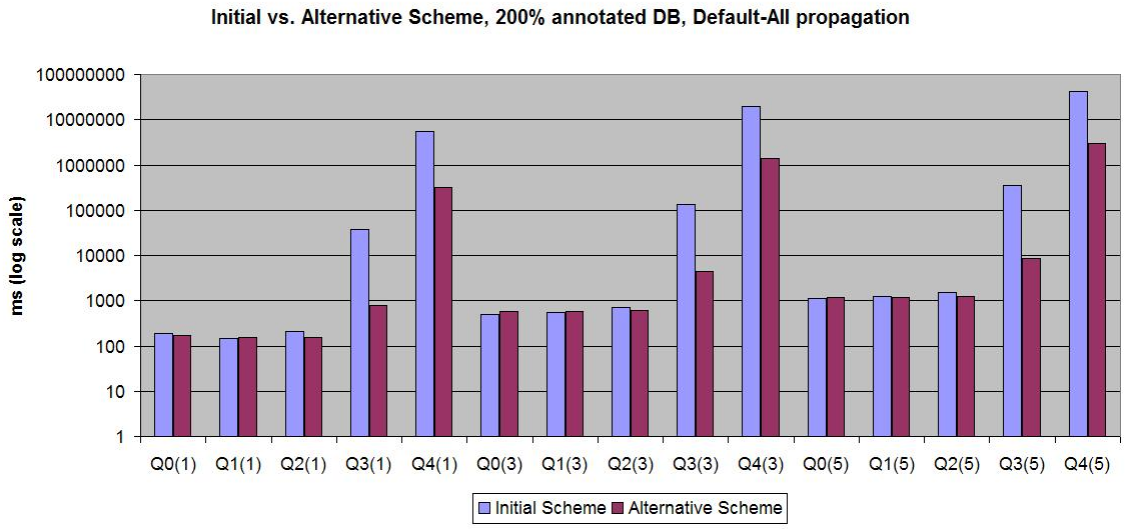



Initial vs. Alternative Scheme, 30% annotated DB, Default-All propagation
ms (log scale)
1000000
100000
10000
1000
100
10
1
Q0(1) Q1(1) Q2(1) Q3(1) Q4(1) Q0(3) Q1(3) Q2(3) Q3(3) Q4(3) Q0(5) Q1(5) Q2(5) Q3(5) Q4(5)
Initial Scheme
Alternative Scheme


Initial vs. Alternative Scheme, 130% annotated DB, Default propagation
ms (log scale)
10000000
1000000
100000
10000
1000
100
10
1
Q0(1) Q1(1) Q2(1) Q3(1) Q4(1) Q0(3) Q1(3) Q2(3) Q3(3) Q4(3) Q0(5) Q1(5) Q2(5) Q3(5) Q4(5)
Initial Scheme
Alternative Scheme


Initial vs. Alternative Scheme, 130% annotated DB, Default-All propagation
ms (log scale)
100000000
10000000
1000000
100000
10000
1000
100
10
1
Q0(1) Q1(1) Q2(1) Q3(1) Q4(1) Q0(3) Q1(3) Q2(3) Q3(3) Q4(3) Q0(5) Q1(5) Q2(5) Q3(5) Q4(5)
Initial Scheme
Alternative Scheme


Initial vs. Alternative Scheme, 200% annotated DB, Default propagation
ms (log scale)
10000000
1000000
100000
10000
1000
100
10
1
Q0(1) Q1(1) Q2(1) Q3(1) Q4(1) Q0(3) Q1(3) Q2(3) Q3(3) Q4(3) Q0(5) Q1(5) Q2(5) Q3(5) Q4(5)
Initial Scheme
Alternative Scheme



In all the above graphs, we observe an important improvement of the execution times between the alternative and the initial storage scheme (note the logarithmic scale of the graphs). In some of the cases, the execution time decreased by a factor of 10.

For some of the very simple queries the execution times appear to be greater for the alternative scheme. The execution times for these queries are very small (generally less than a tenth of a second) and external influences such as slight modifications in the experimental environment (operating system, database server) have a great effect on the experimental value. Thus the comparison for these queries is not very relevant.

The main justification for the improvement in execution times provided by the alternative storage scheme is the following. Under the initial storage scheme, the annotations are stored in the same relation as the data. Thus extracting annotations together with data from one relation is a relatively cheap operation (no joins are necessary). On the other hand, propagating annotations in queries involving several joins between relations becomes a very expensive operation (the joins are performed between large tables with redundancies generated by the way the annotations are stored). In the alternative storage scheme, the annotations are stored in a separate relation. The annotations are extracted by joining the data relation and the annotation relation. The joins required by the query (excluding the ones needed for obtaining the annotations) are performed between much smaller tables (containing only the data, not the annotations in a redundant format). Thus the execution times for queries involving several joins are significantly improved when using the alternative storage scheme.

5 Further remarks

While working on the design and implementation of the alternative storage scheme for DBNotes, I also worked on a data exchange project. At the moment of writing this report, it is still in the development phase, this being the reason why only a few details about it are presented here.

Data exchange deals with transforming a database instance structured under a source schema into an instance structured under a target schema, while satisfying the constraints between the source and target schemas. A data exchange setting is a tuple $(S, T, \Sigma_{st}, \Sigma_t)$ where S is the source schema, T the target schema, Σ_{st} a set of constraints between S and T and Σ_t a set of constraints on T. The data exchange problem is the following: given an instance I valid against S, find an instance J valid against T such that the constraints in Σ_{st} and Σ_t are satisfied.

The goal of the project is to generate synthetic data exchange settings that can be used as a benchmark for data exchange tools. The user can influence the way these settings are generated by providing, at several stages in the generation process, sets of control parameters.

6 Conclusions

This report presented the DBNotes annotation management system, with an emphasis on my main contributions in the project: the design and implementation of an alternative annotation storage scheme, together with the experimental evaluation of the performance of this alternative scheme.

This annotation system uses an extension of SQL, called pSQL, that allows the specification of the way annotations are propagated when data is transformed by queries. This language was presented in the first sections of the report, together with the architecture of the implemented system. Because it is built upon a relational database management system, I had to implement a translation mechanism (presented in a subsequent section) from pSQL to SQL specific to the alternative annotation storage scheme. This scheme had to be evaluated, so I carried out several series of experiments that on one hand show the behavior of the scheme when some parameters change (such as dataset size, number of annotations and type of propagation) and on the other hand compare it to the scheme that had been initially implemented.

The main conclusion that can be expressed after the analysis of the experimental results is that the alternative scheme is more efficient than the initial scheme. The performance gap is most obvious for queries with multiple joins. In our experimental setting, the execution times for queries with three and four joins decreased when using the alternative scheme, compared to the initial scheme, by a factor of about 10.

The period I spent working on DBNotes allowed me to confront very different and interesting problems. I had the chance to be part of a research team at the University of California, Santa Cruz. It was a great opportunity to work in an exceptional academic environment, and was overall an outstanding experience from every point of view.

References

[1] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, Gaurav Vijayvargiya: "An Annotation Management System for Relational Databases", VLDB Journal

[2] Serge Abiteboul, Richard Hull, Victor Vianu: "Foundations of Databases"

[3] Raghu Ramakrishnan, Johannes Gehrke: "Database Management Systems"

[4] Laura Chiticariu, Wang-Chiew Tan, Gaurav Vijayvargiya: "DBNotes: A Post-it System for Relational Databases Based on Provenance", SIGMOD 2005 (Demo track)

[5] TPC Transaction Processing Performance Council, http://www.tpc.org

[6] Wang-Chiew Tan: "Containment of Relational Queries with Annotation Propagation", DBPL 2003

[7] The University of California, Santa Cruz Website, http://www.ucsc.edu