Improv: Flexible Data Provenance for Relational Databases

Nicholas Hunt   Peter Hornyack
{nhunt,pjh}@cs.washington.edu

1. INTRODUCTION

Curated databases, which consist of data extracted from original sources, printed articles, and other databases, are a valuable source of data for scientists. However, as curated databases aggregate information from multiple sources, the origin of the data elements can be lost. Because of this, curated databases often provide support for data *annotations*, which are pieces of extra information added to elements by the human curators in the data extraction and collection process, containing miscellaneous data such as the original data source. However, manually recording the origin of each data element is a tedious and error-prone task. Ideally, the system would provide an automated way to track *data provenance*, or the movement of a data element from its origin to its final place of use.

To help address this problem, we have implemented a mechanism for automating provenance tracking for relational databases. Our system is based upon the Annotation Management System (AMS) described by Bhagwat et al. [4], which uses annotations to track the origin of data elements in a database. In addition to implementing one of the annotation propagation algorithms used in this previous work, we have also added support for propagating annotations to joined and aggregate values. Our implementation, called Improv, operates by transparently transforming SQL queries submitted by users to perform the annotation propagation, without requiring changes to the database implementation itself. This design allows our system to be easily integrated into any application, regardless of the underlying database engine. It is this feature that gives our system its name: the Invisible Mechanism for Provenance.

Prior work has explored several aspects of data provenance, including different annotation models for tracking different types of provenance; *where-provenance*, *why-provenance* and *how-provenance* are examples. Buneman et al. [5] provides an overview of these areas. Briefly, for a particular tuple found in a view or the output of a query, where-provenance allows the database user to determine the base table that contains that tuple; why-provenance tells the user all of the tuples in the database that "contributed to" the output tuple; and how-provenance allows the user to reconstruct a query equivalent to the one that produced the output. For Improv, we chose to track why-provenance, because it is generally more informative than where-provenance, while being simpler to implement than how-provenance.

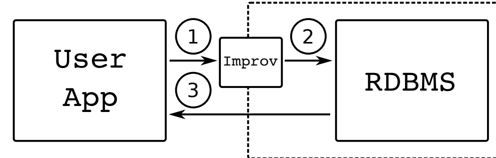


Figure 1: The provenance mechanism sits transparently between the RDBMS and the user application. Unmodified SQL queries from application (1) get translated to annotation-propagating SQL (2); the RDBMS executes this transformed query and returns the results directly to the application (3).

2. DESIGN

2.1 Design Goals

When designing Improv, we had several goals in mind:

1. Provenance should be tracked at the cell (attribute) level.
2. Provenance annotations should be propagated automatically, without any special consideration from the SQL user.
3. The system should expose to applications the same SQL interface that they are used to.
4. The system should be flexible enough to be integrated with any relational database.

The first decision we made was the granularity at which to track data provenance. Some of the possible choices include the relation level, tuple level, or individual cell level. We decided Improv would track provenance at the granularity of individual cells, since it provides the most complete provenance information. As an example of where provenance information can be lost if tracking were performed at the relation or tuple level, consider a simple join of two input relations: with table or tuple level annotations, the precise information regarding which attributes came from which tables would be lost. Although the storage overhead for cell annotations is greater than for tuple or table annotations, prior work suggests that such fine-grained annotation tracking is not prohibitively expensive [4].

Design goals 2 and 3 stem from the desire to create a system that is transparent to the database user. In other words, applications or users do not need to be aware that annotations are being tracked by the system; they should be

able to query the database using the same queries that they have used in the past, and the data returned by our system should not require special parsing or handling to deal with annotations. However, some applications may be interested in analyzing the provenance information, and thus there must be a way for a provenance-aware application to retrieve the annotations for the data as well.

Finally, the purpose of goal 4 is to ensure that our implementation is as usable as possible, without requiring the installation of a custom or modified database system. We wanted to ease the burden of deploying our provenance mechanism as much as possible. We decided to implement Improv as a wrapper around an existing RDBMS, as shown in Figure 1. Ideally, Improv would expose to the user or application the same interface provided by the original database. This design makes Improv "invisible" to both the user and the database.

2.2 Annotation Management System

To meet these design goals, we chose to base our system on the Annotation Management System, an early implementation of a system for tracking where-provenance [4]. AMS has several useful properties, but also some limitations. However, because AMS tracks provenance annotations at the cell granularity, it satisfies the first of our design goals. In Improv we adjust the format of the annotations and the operations that merge them together to track why-provenance, rather than where-provenance.

AMS takes as input its own "pSQL" query language, which consists of the Select-Project-Join-Union subset of SQL (conjunctive queries with union) with an additional clause, `PROPAGATE`, that explicitly specifies how annotations should be propagated through the system. AMS works by storing a column of annotations along with each normal data column in a table; pSQL queries are then "translated" into standard SQL queries that propagate the annotations at the same time that the query is evaluated. Because this translation and propagation is hidden from the user, it fits our goal of automatically propagating annotations within the database.

2.2.1 Limitations of AMS

One downside of the AMS design is that equivalent SQL queries can produce slightly different annotation outputs. To address this issue, AMS uses two different algorithms for propagating annotations: a `default` algorithm that ignores the issue, and a `default-all` algorithm that propagates annotations in a way that captures all possible equivalent queries. Improv currently only supports the `default` algorithm, because it is simpler to implement and because it is not clear that users always require the semantics of `default-all`. However, `default` as described in AMS does not support annotation propagation for joins, so we add this support in Improv.

Update and delete operations are not cleanly compatible with the AMS approach of propagating annotations concurrently with query execution, so AMS does not support these operations, and neither does Improv. This results in an "append-only" database model, which matches some databases that are used today [2].

Another limitation of AMS is the lack of support for aggregates and bag semantics. One of the key features of our implementation is the support for aggregation operators that we have added (see Section 3.1). Our system currently does not support bag semantics, but Section 3.5 describes one possible way that bag semantics could be added.

3. IMPLEMENTATION

Improv is implemented as a wrapper around a conventional relational database management system. We have implemented Improv in Java and provide a simple API that application programmers can use to connect to database servers and execute SQL queries. We are currently using PostgreSQL as the backing database, but because Improv uses a JDBC interface for database communication, any database backend supported by JDBC is also supported by Improv.

Referring again to Figure 1, Improv intercepts all queries sent to the database by applications or database users and transforms those queries into expanded SQL queries that propagate provenance annotations. These expanded queries are then passed to the backing RDBMS and the results are returned to the caller; the caller can configure an Improv setting to choose whether or not to see the annotations in the query results. The transformation of specific query types are described in Sections 3.3, 3.4, and 3.5 below.

3.1 Attribute Annotations

Shadow Attributes. We track data provenance for each cell in a tuple. To support these annotations, we allocate *shadow attributes* for each attribute in a relation. That is, for every attribute x in a relation, there is a corresponding attribute x_a that tracks the provenance for x. Initially creating these shadow attributes is discussed in detail in Section 3.3.

For our purposes, we are interested in collecting *sets* of annotations. When two or more tuples contribute to the value of a cell in a relation, that cell's annotations should be the union of all the annotation sets for the contributing cells. Most database engines do not provide a set type, however, and thus we represent sets as a simple comma-separated list of set elements, stored in a standard `TEXT` column type.

Annotation aggregation. For our implementation of annotation propagation, we frequently require the use of an SQL aggregate function that performs a set union over a collection of tuples. Because PostgreSQL does not support such an aggregate by default, we implemented our own user-defined function, `setunion`, that performs this required operation. Using a user-defined function may limit our portability to other database engines, although most RDBMS today provide this support.

Because our annotation sets are represented as a list of set members, `setunion` is implemented by iterating over each tuple annotation list in the aggregate, and maintaining an in-memory hash table of the unique values seen so far in any annotation set. This hash-table can then be iterated to produce the union in $O(n)$ time, where n is the number of tuples in the relation being aggregated, and in $O(m)$ space, where m is the number of unique values in the attribute being aggregated.

3.2 SQL Parsing

Parsing the SQL expressions in Improv is handled by the open-source SQL parser ZQL [1]. Although ZQL supports most common SQL constructs, it has a number of limitations. For instance, ZQL is unable to understand nested

```
-- Before transformation
CREATE TABLE cust (
    id   INTEGER,
    name VARCHAR(32));

-- After provenance transformation
CREATE TABLE cust (
    id     INTEGER,
    id_a   TEXT,         -- Provenance for id
    name   VARCHAR(32),
    name_a TEXT);        -- Provenance for name
```

Figure 2: Transforming a CREATE statement. Each column gets a new annotation column to track the provenance tags.

queries in the FROM clause of an SQL SELECT statement, and Improv inherits this limitation. Also, ZQL is unable to parse CREATE statements, but we were able to extend the parser to support these.

3.3 CREATE Statements

When a user creates a new relation using a CREATE statement, Improv needs to transparently create the shadow attributes described above to contain the provenance annotations. Creating these additional shadow attributes is done by iterating over each of the columns specified in the original CREATE, and for each column, appending an additional attribute to the schema. The name of this new attribute is the same as the original, with an additional suffix of "_a" to designate it as an annotation. We assume for simplicity that these new column names do not conflict with column names already present in the user's original input query.

As an example of a CREATE transformation, Figure 2 shows the creation of a simple customer table cust containing a customer ID and name. Our transformation includes the same two attributes as the original query, but also includes the shadow attributes for holding provenance information.

3.4 INSERT Statements

Insertion into a database is performed with an INSERT statement, indicating the values to insert and the columns to insert them into. Transformation of INSERT statements is performed similarly to the CREATE statements discussed above: Improv will iterate over the values specified in the original query, and construct a new query identical to the first with additional values specified for the annotation attributes. Figure 3 shows the result of a transformation of a query inserting a new record into the cust relation defined above.

```
-- Before transformation
INSERT INTO cust (id, name)
       VALUES    (10, 'Joe');

-- After provenance transformation
--    ann1 and ann2 are defined in the main text
INSERT INTO cust (id, id_a, name, name_a)
       VALUES    (10, ann1, 'Joe', ann2);
```

Figure 3: Transforming an INSERT statement. The query is modified to include initial values for the corresponding shadow attributes.

The initial annotations assigned to cells in the database are globally unique values. Currently, the annotations are strings consisting of the username of the user who is inserting the value, the name of the table the value is being inserted into, and an integer making the entire annotation unique. For instance, if the user nhunt were to insert a new value into the cust relation, an annotation for the id column could be nhunt:cust:152. The values ann1 and ann2 in Figure 3 would be similar to this previous example.

An alternate implementation might just be concerned about which database users contribute to a value, and thus might opt for just including the username. With just the username annotations, it would be possible to determine which users have contributed data that affected the value of the cell under investigation, but precise information about exactly what data contributed to its value would be lost. We chose our scheme to provide the maximum level of detail, but this choice can easily be configured based on the needs of the application.

3.5 SELECT Statements

SELECT statements are the workhorse of our annotation propagation algorithm. Our goal is to transform SELECT statements such that each output column has a corresponding annotation column that contains the union of the annotations for every column in the relation that contributed to each cell's output value.

Consider, for example, the simple SELECT statement in Figure 4. Before applying our transformation algorithm, the query simply projects the id and name attributes from the cust relation. Thus, in addition to the id and name attributes, our transformed query must also project the corresponding annotation columns. These additional projections can be seen in the inner query of the transformed statement.

```
-- Before transformation
SELECT id, name FROM cust c;

-- After provenance transformation
SELECT id, name,
       setunion(id_a) AS id_a,
       setunion(name_a) AS name_a
FROM   ((SELECT c.id, c.name,
                c.id_a AS id_a,
                c.name_a AS name_a
         FROM   cust c)
) AS psqlResult GROUP BY id, name
```

Figure 4: Transforming a simple SELECT statement. The inner query in the FROM clause projects the annotations and the outer SELECT aggregates all annotations for the project attributes.

In this example, each output attribute only gets data from a single input attribute, and thus the output annotations are a simple projection of the input annotations. For queries that combine data from two or more columns in the input relations, the annotations for all contributing cells must be reflected in the final annotation of the output relation. As a more complex example to illustrate this, consider the simple equijoin operation shown in Figure 5, where the cust and sales relations are joined on the customer id.

The first sub-query in the FROM clause projects the desired application-visible attributes for the id, name and price, in

```
-- Before transformation
SELECT c.id, c.name, s.price
FROM   cust c, sales s
WHERE  c.id = s.cid;

-- After provenance transformation
SELECT id, name, price,
       setunion(id_a) AS id_a,
       setunion(name_a) AS name_a,
       setunion(price_a) AS price_a
FROM   ((SELECT c.id, c.name, s.price,
                s.price_a AS price_a,
                c.name_a AS name_a,
                -- Project annotations from c
                c.id_a AS id_a
         FROM   cust c, sales s
         WHERE  (c.id = s.cid))
        UNION
        (SELECT c.id, c.name, s.price,
                NULL AS price_a,
                NULL AS name_a,
                -- Project annotations from s
                s.cid_a AS id_a
         FROM   cust c, sales s
         WHERE  (c.id = s.cid))
) AS psqlResult GROUP BY id, name, price
```

Figure 5: Transforming an equijoin `SELECT` statement. The annotations for both the `s.cid` and `c.id` attributes are projected to the final `id` attribute in the output relation.

addition to the annotations for these attributes. However, because the `id` in the output relation is the result of an equijoin with the `cid` attribute of the `sales` relation, the second query of the inner union projects the annotations for `s.cid` to the output annotations for `id` as well. The outer query will then group by the attributes projected by the original query, and aggregate the annotation columns using `setunion`, the custom set aggregator described in Section 3.1.

Because of the `GROUP BY` in the transformed query, our current implementation does not support the bag semantics of standard SQL. For instance, projecting a non-key column can produce duplicates in standard SQL, but our system would eliminate duplicates as a result of the grouping. Although not examined in detail for this project, we could continue to support bag semantics by adding a "multiplicity" annotation to the tuples, indicating the number of times this tuple would have appeared in unmodified SQL.

3.5.1 Aggregates and Duplicate Elimination

Aggregate functions (such as `MIN`, `COUNT`, `AVG`, etc.), and duplicate elimination are handled in our system by propagating the annotations of all cells involved in the aggregate (or the duplicate tuples) to the corresponding annotation attribute of the output relation. For example, in the query `SELECT SUM(price) AS sum FROM sales`, the output relation would have two attributes: one for `sum` and one for the annotations for `sum`. The `sum` attribute would contain the sum of all values in the price column and the annotation attribute would be the union of all the annotations of those cells.

4. EVALUATION

We evaluate our system both objectively and subjectively. Our objective evaluation aims to determine Improv's performance overhead and scalability for different types of database queries. Our subjective analysis will provide a brief retrospective on our experiences using the system.

4.1 Methodology

Our performance evaluation was performed on a 4 core Intel Xeon server with 2 GB of RAM. All of the experiments were executed 5 times, discarding the minimum and maximum times, and the mean and 95% confidence interval of the remaining three trials is presented in the data below. We tried to minimize the amount of background processes running on the system, but because our test server is a shared system, there was inevitably noise introduced in our results by other users on the system. However, given the execution time of our tests and the relatively tight grouping of the individual trials, we feel this noise is minimal and thus did not have a significant effect on our results. Both the database server and the client were run on the same machine to reduce the perturbation of our results due to network latency.

4.1.1 System Configurations

To evaluate our system, we fixed a set of four different types of SQL queries, and measured the time required to execute those queries under four different configurations. The configurations shown in the results are:

Native SQL Queries with Unannotated Tables This provides a baseline for how the SQL query performs on the unmodified database. The tables do not contain the shadow annotation columns, and the queries do not propagate provenance.

Native SQL Queries with Annotated Tables The tables contain the shadow attributes for annotations, but the queries are not translated by Improv and thus do not propagate the annotations to the output.

Improv Queries with NULL Propagation Tables have annotations and all queries are translated to propagate annotations; however, the custom `setunion` annotation aggregation function is replaced with a NULL aggregator.

Improv Queries with Full Propagation This is the full implementation of our provenance mechanism. All relations have shadow attributes for annotations and all queries are correctly translated to propagate the annotations to the output, as described in the preceding sections.

This set of configurations was chosen to isolate various components of our implementation to help understand where some of the system overheads come from. Comparing the first and second configurations allows us to determine the performance overhead due to just the presence of the provenance annotations in the base tables. Comparing the second and third configurations will show how much additional overhead is introduced by executing the transformed query that performs the annotation propagation; as shown in Figure 5, this transformed query is often significantly more complex than the original query. Finally, comparing the third and fourth configurations shows the overhead of our

```
SELECT ndb_no, nutr_val, min_val, max_val
FROM   nut_data
WHERE  nutr_val > 0.5;
```

(a) Simple selection/projection query used for evaluation

```
SELECT f.ndb_no, f.shrt_desc, n.nutrdesc, n.nutr_val
FROM   nutrients n, foods f
WHERE  n.ndb_no = f.ndb_no;
```

(b) Join query used for evaluation

```
SELECT nutr_no, COUNT(nutr_val) AS c,
       SUM(nutr_val) AS s
FROM   nutrients
GROUP  BY nutr_no;
```

(c) Aggregate query with few bins, many elements per bin

```
SELECT nutr_val, COUNT(nutr_no) AS c,
       MIN(nutr_no) AS m
FROM   nutrients
GROUP  BY nutr_val;
```

(d) Aggregate query with many bins, few elements per bin

Figure 6: SQL queries used to evaluate Improv

setunion aggregate operator and the cost of maintaining the annotation sets.

4.1.2 Query Types

For each of the system configurations given in the previous section we evaluate four different SQL queries and measure the time required to perform the operation as a function of database size. The four queries were chosen to evaluate our system under a variety of workloads. The queries chosen were as follows:

Simple Selection/Projection. The query for selection and projection simply projects a set of attributes for tuples matching a particular selection criterion in a single relation; it is shown in Figure 6(a). One of the projected columns is a key, so no duplicate elimination happens with this query.

Simple Join. This query performs an equijoin on two relations in the database and projects a set of the attributes. Figure 6(b) shows the query used for this case.

Aggregates/Duplicate Elimination. We look at two queries that perform aggregation and duplicate elimination. The first query, show in Figure 6(c), groups by an attribute that has a relatively small number of unique values; thus, each group has a large number of elements assigned to it. The second aggregate query, shown in Figure 6(d), groups by an attribute with a large number of unique values, resulting in a large number of groups and a small number of tuples aggregated in each group.

4.1.3 Test Data Sets

For our evaluation, we used a subset of the National Nutrient Database for Standard Reference [3], provided by the USDA. The tables used in our evaluation along with the number of tuples in each are shown in Table 1. The first

Relation	No. of Tuples
food_des	7104
nut_data	555726
nutr_def	146
foods	7104
nutrients	555726

Table 1: Size of the relations used in our evaluation

group of relations are base tables provided by the data set. The second group contains views we created on this data: foods is a simple projection of a subset of the attributes in food_des, whereas nutrients is a join of the nut_data and nutr_def relations. To vary the size of the database for our evaluation, for each of these tables we created multiple smaller tables by randomly choosing a percentage of the tuples to keep.

4.2 Experimental Results

4.2.1 Comparison of Improv to native database

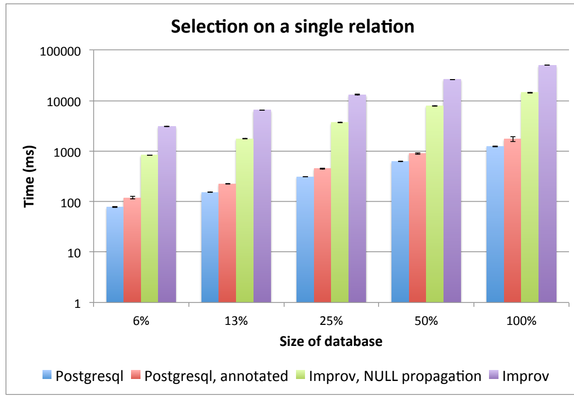
The performance and scalability results are shown in Figure 7. The graphs in this figure show the wall-clock time measured while executing each of the four evaluation queries. Within each cluster, the different bars show the runtime for each of the four system configurations. The different clusters within a graph show the effect database size has on the query runtime. The missing bars in Figure 7(d) for Improv are due to the execution time being too long.

There are several points to take away from the graphs in Figure 7. First, in all cases, comparing native PostgreSQL with and without annotations shows that the presence of the annotations in the databases has a small but noticeable performance impact. This is likely because even though the native PostgreSQL queries that we run do not explicitly touch the shadow annotation columns, these columns may be read in from disk or may occupy some memory before they are projected away in the query execution.

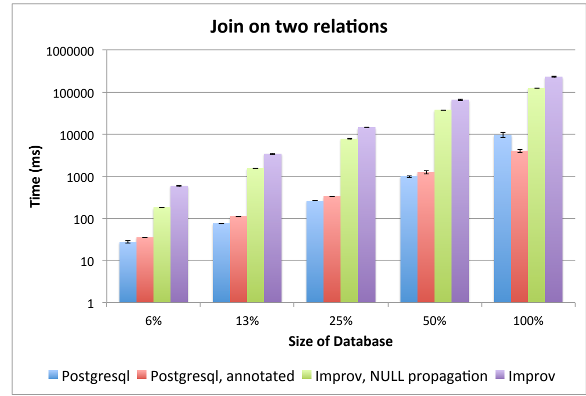
Comparing native PostgreSQL to Improv shows that Improv introduces significant performance overhead, even when NULL propagation is used. Improv executes simple select and join queries about 12x more slowly than native PostgreSQL; for queries with aggregation, Improv is more than 100x slower. For select and join queries, this performance overhead is due almost entirely to the increased complexity of the transformed query that propagates the annotations: the simple select query is transformed into a query with one subquery in the FROM clause, while the simple join query becomes a query with a union of two subqueries in the FROM clause. Our timing instrumentation indicated that the time to transform a query before it is evaluated is negligible.

Despite the large constant performance overhead that Improv has for select and join queries, it does preserve scalability for these queries. For the select query, the difference in execution time for the smallest and largest database sizes is slightly greater than one order of magnitude, for both native PostgreSQL and Improv; this is seen similarly for the join query. However, this is not the case for the aggregate queries, which scale worse for Improv than they do for native PostgreSQL.

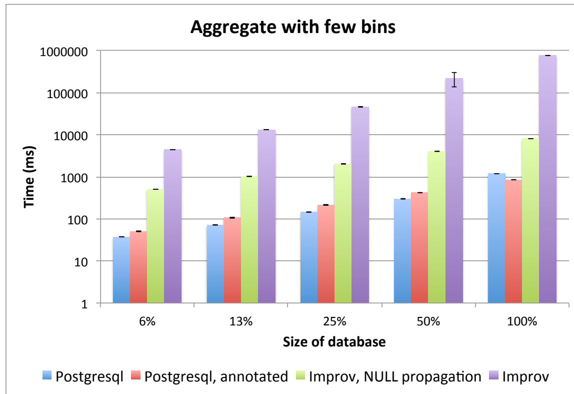
Aggregate operations perform so poorly in Improv because of our choice of annotations. In Figures 7(c) and 7(d), com-



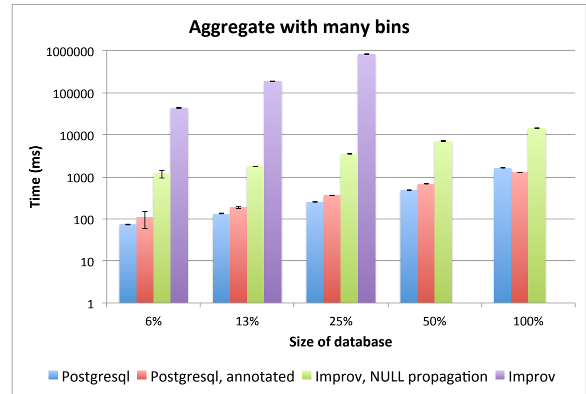
(a) Simple selection and projection



(b) Join between two relations



(c) Aggregate with few bins and many elements



(d) Aggregate with many bins and few elements

Figure 7: Performance and scalability results for each of the query types and system configurations. Both axes use a logarithmic scale.

paring the execution times of Improv with NULL annotation propagation and Improv with full propagation shows the cost of using our `setunion` operator to keep sets of strings for the annotations. As the database size increases, both the number of "bins" for the values to be grouped by and the number of tuples in each bin grows; computing an aggregate means that all of the annotations for the cells of the tuples in a bin must be unioned together with `setunion`. Our choice to use `TEXT` attributes for the annotations, and our resulting implementation of `setunion` that involves string concatenation and hash table manipulations in Perl, makes unioning together all of these annotations a very expensive operation which does not scale linearly with the size of the database. The difference between Figure 7(c) and Figure 7(d) suggests that there is some fixed cost for computing the `setunion` for each bin (perhaps for constructing and traversing a hash table) that is independent of the number of annotations in that bin, which causes the number of bins to be a greater factor in performance cost than the number of annotations in each bin.

These results clearly show that by improving the way that Improv tracks annotations, our system could see significant performance improvements for queries with extensive grouping or duplicate elimination. An alternate implementation might keep a separate one-to-many relation containing the annotations for the cells: each cell in the database would be assigned a globally unique integer identifier, and this identifier can be used to look up the annotations in the annotation relation. This would replace the expensive string operations with the standard insertion operations that are highly optimized.

4.2.2 *Performance across query types*

Figure 8 shows a direct comparison of the runtimes for the different types of fully annotated queries with Improv, as the size of the database grows. As expected, the simple selection query is by far the most performant since no annotations need to be merged to satisfy the query. The join is the second fastest and the two aggregate queries perform significantly worse. The aggregation with many bins very quickly becomes impractical to compute.

4.3 Usability Evaluation

Our system was designed to allow the user to determine which cells in a relational database contributed to the current value of a particular cell, and to do this in a transparent way. From this perspective, we feel our system works well. Users can submit queries, and the results are all annotated with the provenance tags, and the tags provide enough information to the user to determine the source. Figure 9 shows an example of the annotation returned by Improv for a cell that was used to join two relations. It is clear that the cell with this annotation has its current value because of data inserted into the database from two different users, `sue` and

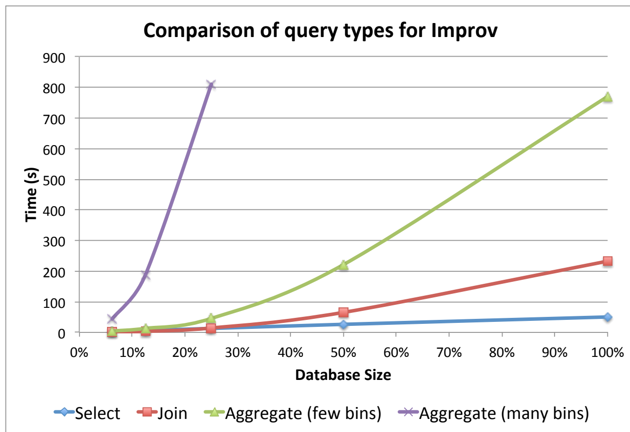


Figure 8: Comparison of overheads for different query types with fixed database size and full annotation propagation.

pjh, and that the data came from two different relations, food and nutrition.

```
sue:foods:5244,pjh:nutrition:1989
```

Figure 9: Example annotation for a cell used to join two relations.

While the join annotations are fairly easy for the user to understand, we have found the annotations for aggregates to be slightly more cumbersome. Consider the annotation in Figure 10, which is the annotation for a cell that is the aggregate of 8 tuples.

```
pjh:nutrition:7705,pjh:nutrition:5925,
pjh:nutrition:6317,pjh:nutrition:2079,
pjh:nutrition:7993,pjh:nutrition:21,
pjh:nutrition:9457,pjh:nutrition:2080
```

Figure 10: Example annotation for an aggregate cell.

Because we aggregate all annotations of cells involved in the union, the length of the annotation will grow linearly with the number of cells being aggregated. While the annotation allows the user to determine exactly what cells contributed to the aggregate value, the length of the annotation may make it difficult to understand. This example may be a case where a less precise annotation may be more meaningful. For example, maybe for aggregates the annotation can be identify the attribute being aggregated, rather than the individual cell in every tuple. Although less precise, this may be more useful for a user trying to understand the output of our system.

5. RELATED WORK

Prior to AMS, other work had described where- and why-provenance and proposed some mechanisms for tracking it. Buneman, Khanna, and Tan [6] introduced the concept of where-provenance as distinct from why-provenance, and presented a method where, given a query and a database, a "reverse query" can be generated that determines the provenance for a single output tuple. This reverse query technique was also demonstrated by Cui, Widom and Wiener [7] for tracking why-provenance. However, if provenance information is desired for a large set of output tuples, a large number of these reverse queries must be generated and evaluated, so AMS and Improv take the approach of propagating annotations within the database on every query and view creation.

Green et al. [8] showed that when annotated relations are expressed as a *commutative semiring*, a positive closed algebra exists that propagates the annotations across sequences of relational operations. Annotating input tuples with unique ids and using a simple union operator in the algebra expresses why-provenance: the set of input tuples that contributed to an output tuple. Furthermore, how-provenance can be expressed by using sum and product operators, rather than union, and by using polynomials as symbolic representations of semirings. How-provenance is a superset of why-provenance that describes (unlike a set) *all* of the possible ways that an output tuple could have been constructed from the input tuples. Other types of annotations and operators on commutative semirings can express other properties of relations, such as bag semantics; Improv could be adapted to propagate these other types of annotations in order to track other useful information instead of (or perhaps in addition to) why-provenance.

6. CONCLUSIONS

Improv improves upon the AMS `default` algorithm for propagating annotations by adding support for join and aggregate operations. Our system succeeds in tracking the why-provenance of each cell in the database, although we find that why-provenance may not represent exactly the information that a user desires for certain operations, such as aggregation. The performance overhead of Improv for tracking provenance is significant. We find that much of the performance cost could be eliminated by optimizing parts of our implementation, particularly the annotation format and the operator used to merge annotations together. Improv succeeds in remaining "invisible" to database clients and servers, requiring only the addition of a custom aggregator function to the database server in our current implementation.

7. REFERENCES

[1] ZQL: a Java SQL Parser, September 2010. `http://zql.sourceforge.net/`.

[2] SQLShare: Database-as-a-Service for Researchers, February 2011. `http://escience.washington.edu/sqlshare`.

[3] USDA National Nutrient Database for Standard Reference, March 2011. `http://www.ars.usda.gov/Services/docs.htm?docid=8964`.

[4] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 900–911. Morgan Kaufmann, 2004.

[5] Peter Buneman, James Cheney, Wang Chiew Tan, and Stijn Vansummeren. Curated databases. In Maurizio Lenzerini and Domenico Lembo, editors, *PODS*, pages 1–12. ACM, 2008.

[6] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.

[7] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[8] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In Leonid Libkin, editor, *PODS*, pages 31–40. ACM, 2007.