A Spreadsheet Algebra for a Direct Data Manipulation Query Interface

Bin Liu [1], H.V. Jagadish [2]

Department of EECS, University of Michigan
Ann Arbor, USA
[1]binliu@umich.edu
[2]jag@umich.edu

***Abstract*— A spreadsheet-like "direct manipulation" interface is more intuitive for many non-technical database users compared to traditional alternatives, such as visual query builders. The construction of such a direct manipulation interface may appear straightforward, but there are some significant challenges. First, individual direct manipulation operations cannot be too complex, so expressive power has to be achieved through composing (long) sequences of small operations. Second, all intermediate results are visible to the user, so grouping and ordering are material after every small step. Third, users often find the need to modify previously specified queries. Since manipulations are specified one step at a time, there is no actual query expression to modify. Suitable means must be provided to address this need. Fourth, the order in which manipulations are performed by the user should not affect the results obtained, to avoid user confusion.**

We address the aforementioned challenges by designing a new spreadsheet algebra that: i) operates on recursively grouped multi-sets, ii) contains a selectively designed set of operators capable of expressing at least all single-block SQL queries and can be intuitively implemented in a spreadsheet, iii) enables query modification by the notion of modifiable *query state*, and iv) requires no ordering in unary data manipulation operators since they are all designed to commute. We built a prototype implementation of the spreadsheet algebra and show, through user studies with non-technical subjects, that the resultant query interface is easier to use than a standard commercial visual query builder.

I. Introduction

A. Motivation

Non-technical users find it challenging to specify queries against structured databases. This is true even with visual query builders, which provide a largely "point-and-click" means to develop query specifications. One possible reason for this is the separation of query specification from result presentation in most databases available today. The theory is that humans are good at manipulating things that they can "touch", but it requires substantially greater technical sophistication to be able to abstract the specification into a query that must be fully specified separate from the data being operated on before it can be executed.

To address this need, the term direct manipulation was coined by Shneiderman [1], [2], [3] to refer to systems that support: i) continuous representation of the object of interest, ii) physical actions or labeled button presses instead of complex syntax, and iii) rapid incremental reversible operations whose impact on the object of interest is immediately visible.

In the text editing context, systems with similar properties have been called *WYSIWYG* (What You See Is What You Get). In the database context, with a direct manipulation interface the user always has on hand some data set, currently being analyzed or manipulated. A chunk of the data set is visible on the screen – all of it is not likely to fit, except for the smallest data sets. Initially, the current data set may be source data – say a relation in a database. After each manipulation is performed, the user has intermediate result data available. Eventually, the final results become available to the user. Modifications to the data set at hand are performed by "directly" specifying operators to be applied to it. To meet the second and third desiderata above, each such user-specified manipulation must be fairly simple, and the result of applying it reflected immediately by updating the current data set.

Spreadsheets (for example, Microsoft Excel and OpenOffice Calc) are popular means for analyzing data through direct manipulation. They are frequently used to analyze data extracted from database systems, particularly in the context of decision support. However, spreadsheets are not designed for querying databases (although it is possible to use them to load data from a database). Our objective is to create a spreadsheet-like interface to directly query and access relational databases through direct manipulation. To accomplish this we need to overcome several challenges, which we describe next, beginning with an example.

B. Conceptual Challenges

Sam, a new graduate student at the University of Michigan, is looking for a sedan in a used car database, which maintains common attributes of cars (e.g., year, model, mileage, price, condition). Some sample records are shown in Table I.

Query Division Challenge Sam is interested in late model cars (2005 or later) in good or excellent condition, and he would like the results grouped by Model and ordered by Price. This is a simple query to specify in SQL. But, on a spreadsheet, just by pointing and clicking, there is no straightforward way to state all these requirements at once. Instead, Sam has to break down his need into parts, and specify one part at a time (e.g. "select late model cars"). Upon asking for this, Sam will immediately see all late model cars, irrespective of condition, and not grouped and ordered as he would like, since these requests have not yet been made.

TABLE I

SAMPLE USED CAR DATABASE

ID	Model	Price	Year	Mileage	Condition
304	Jetta	$14,500	2005	76,000	Good
872	Jetta	$15,000	2005	50,000	Excellent
901	Jetta	$16,000	2005	40,000	Excellent
423	Jetta	$17,000	2006	42,000	Good
723	Jetta	$17,500	2006	39,000	Excellent
725	Jetta	$18,000	2006	30,000	Excellent
132	Civic	$13,500	2005	86,000	Good
879	Civic	$15,000	2006	68,000	Good
322	Civic	$16,000	2006	73,000	Good

This division of a single query into pieces is not in itself challenging, but it is the root of other difficulties we discuss below.

Grouping Challenge Whenever data is displayed, issues of ordering and grouping must be considered. In relational systems, grouping is "hidden" in that it is paired with aggregation, and ordering is frequently outside the main algebraic query specification, treated as a post-operation for display purposes. Due to query division, we have to worry even about "intermediate results" in the case of a spreadsheet. Since these are shown to the user, we have to be concerned with ordering and grouping even for the results of each intermediate operation. This takes us from the realm of sets (or multi-sets) in the case of relational systems to collections that support grouping and ordering.

Aggregation Challenge Aggregate computation is a common operation invoked during data analysis. However, computation of a relational aggregate query results in the definition of a new relation which is typically not union compatible with the original relation, making it hard to store the data and its aggregates all on a single "spreadsheet." For example, suppose Sam wants to see cars whose price is lower than the average for that Model and Year. It is not immediately obvious how to accomplish this through direct manipulation starting from a display similar to Table I. (A SQL query to accomplish this would involve nesting, and a join between two copies of the base table).

Query Modification Challenge In interactive data analysis, users may often find the need to make small modifications to previously issued queries. For example, upon finding too many results in model"Jetta" in year 2005 and price less than $18k (and meeting many other conditions he specified), Sam may feel optimistic and desire to change the year to 2006. In a direct manipulation interface, complex queries are developed one operator at a time, and a complete query expression is never explicitly articulated. We would like to provide the user with a facility that is the equivalent of making a small modification to a large query expression and re-submitting it for evaluation.

Operator Ordering Challenge A complex query specification may involve the specification of a sequence of multiple operators. A non-technical user may find it strange if applying the same operators in a different sequence produces different results, particularly since there is no query expression showing explicit operator ordering or parentheses. For example, suppose Sam first computed the average price of all cars of each model, and then realizes he is only interested in cars in 2005, he should be able to just specify that condition and the average price should correct itself immediately. In effect, it should be the same as if the two operations were specified with the selection first and the average afterward. If such automatic recomputation is not feasible, there should at least be a suitable notice given to the user so that unintended wrong results are not quietly computed.

C. Contributions

The first contribution of this paper is the development of a spreadsheet algebra introduced in Sec. III. We developed precise semantics of a spreadsheet model and all algebraic operators that manipulate data in the spreadsheet. The unit of manipulation in the algebra is a recursively grouped ordered multi-set of tuples to address the grouping challenge. Operators in this algebra have been carefully designed so that all unary data manipulation operators commute to address the operator ordering challenge.

Spreadsheets support aggregate computation, and permit storage of aggregates in cells with "computed attributes" – the value of such a cell is defined by means of an expression in terms of the values of the cells being aggregated. This is a notion that spreadsheet users are used to and can be expected to exploit readily. Exploiting this notion, aggregation is defined not as an operator directly, but as the creation of a corresponding computed attribute. This addresses the aggregation challenge.

All single block SQL queries with selection, projection, join, grouping, aggregation, group selection (the HAVING clause), and ordering, can be expressed in this spreadsheet algebra. These, and other (e.g., operator commutativity), properties of the spreadsheet algebra are explored in Sec. IV.

The query modification challenge is addressed in Sec V through a novel proposal for re-writing query history by exploiting the *query state* retained in the spreadsheet interface, and exploiting spreadsheet algebra properties discussed in Sec. IV.

The third contribution of this paper is a spreadsheet interface to an RDBMS that implements the spreadsheet algebra. The core of this interface is the design of specific implementations for spreadsheet algebra operators. The design is presented in Sec. VI.

Our final major contribution is an empirical assessment of the spreadsheet interface. We built a prototype, SheetMusiq, with ideas in this paper. Experiments with human subjects not familiar with SQL show that our implementation is easier to work with than a representative popular visual query builder. Specifically, for a wide assortment of queries given to them in English, users were able to express the query and obtain results faster with the spreadsheet presentation than with the visual query builder. Details are given in Sec. VII.

The rest of the paper is organized as follows. We first define the spreadsheet model in Sec. II, and then present the spreadsheet algebra for this model in Sec. III. In Sec. IV, we study the expressive power of the algebra and the commutativity among unary operators, which lays a solid foundation for query modification in Sec. V. We then present a design (Sec. VI) and user evaluation (Sec. VII) of user interface built upon the algebra, before concluding the paper with related work.

II. The Spreadsheet Data Model

A *data model* (as defined in [4]) contains a notation for describing data, and a set of operations used to manipulate that data. For example, in relational data model, we have "relation" and relational algebra operators. We seek to define the notation and structure of the data model in this section, and will consider specific operators in Sec. III.

A. Intuition

The fact that the spreadsheet is used to present data to users for manipulation poses unique requirements not met by the relational data model. Key amongst these is that the data must be grouped and/or ordered after every operator. Furthermore, the user should not have to re-specify the grouping and ordering with each small operation performed. As such, grouping and ordering must be retained through operators, to the extent possible. For example, a selection condition applied to the data should not change its grouping or ordering.

The basic unit of the spreadsheet algebra is a *spreadsheet*. Unlike a relation, which is an unordered set of tuples, a spreadsheet must support both grouping and ordering. A spreadsheet that has no grouping (or ordering) specified is said to be grouped (resp. ordered) by NULL.

Grouping is not in textbook relational algebra [5]. In SQL, it is always associated with aggregation. The aggregation and grouping operator, if treated as a single algebraic operator, is quite heavy weight. It can involve a GROUP BY clause, a HAVING clause, multiple aggregate functions, and constraints between the list of attributes in the SELECT clause and the GROUP BY clause. A single operator that does all this is not within the spirit of direct manipulation. Rather, we seek to define a separate operator for each logical component of this mega-operator – one for grouping, one for each aggregate, and one for each group selection predicate (in the HAVING clause). But this new grouping operator is not closed over relations (grouping is lost in a set), which creates a problem.

In relational algebra with grouping and aggregation, closure is achieved by insisting that attributes not in the grouping list be projected out (after aggregation and group qualification have been performed, if specified), leaving precisely one tuple in place of each group. In the spreadsheet model, we achieve closure by defining the algebra not over relations restricted to sets of tuples but over spreadsheets that are *recursively grouped set of tuples*. Simply put, a recursively grouped set of tuples is a set of tuples with grouping information. If no grouping is initially specified, the spreadsheet is grouped by NULL. When grouping contains multiple *levels* (e.g., group the cars first by Model, then by Year), a recursive grouping is formed. Each level of group is a relational group. We number the levels of group from the outermost – the root (first) is the spreadsheet itself, cars of the same Model form the second level, and the highest (or *finest*) level groups are cars with same Model and Year. The *basis* of a level of grouping is the set of attributes whose values are the same for all tuples inside any group at this level but not in the parent level. Following the example, we say the "basis" for each level of grouping, from outermost, are $\{$NULL$\}$, $\{$Model$\}$, and $\{$Model, Year$\}$. We will often find it convenient to speak of the *relative grouping basis* as the difference between the basis for one level of grouping and level below it. Thus, the innermost level has a relative grouping basis of Year.

Order can be specified inside each level of group. In the finest level of group, tuples can be ordered by any attribute that is not in any grouping basis. For example, we can order cars with the same Model and Year by Price. For other level of groups, the ordering attribute is already specified by the grouping, and thus only "descending" or "ascending" is allowed. Specifically, the ordering attributes are those in the grouping basis of the immediate higher level but not this level. For example, for the second level groups (cars with the same Model but different year), cars are automatically grouped by Year.

Note that any recursive grouping can be emulated by a single ordering specification in that all tuples can be placed in the same order. To accomplish this, specify order by the lowest level group first, then the next level group, and so on, until finally the order within the highest group, with each order being the same as in the recursive grouping to be emulated. Mere ordering does not provide group identification, though, and leaves all tuples in a single set rather than in a set of (set of...) sets. We would then be unable to specify operators that compute any function of groups.

Whereas a spreadsheet could be used to represent data stored with various organizations, relational databases are prevalent today, and so using a spreadsheet to represent a relation is the natural thing to do. In this paper, we restrict ourselves to the use of spreadsheets to represent relational data. Specifically, a single spreadsheet is used to represent a single relation. Each column in the relation corresponds to a column in the spreadsheet. In addition, the spreadsheet may have some *computed columns*. (We will describe the use and importance of these computed columns in Sec. III).

B. Definition of the Spreadsheet Model

Based on previous discussion, we now formally define the spreadsheet data model. Throughout the paper, we use subscripted lowercase letters to denote elements in sets or lists represented by their corresponding capital form. Superscript denotes the version of an object. For example, g_i^0 denotes the i-th element in G^0 ($1 \leq i \leq |G^0|$). If G^0 is changed, we create a new version, G^1, and its elements are now denoted as g_i^1.

Definition 1 (**Spreadsheet**)**.** A *spreadsheet* S is a multi-set of tuples specified by a quadruple (R, C, G, O), where

1) R is a reference to the relation it represents, known as the *base relation* of S.
2) C is a superset of columns in R,
3) G is a list for grouping specification. Element g_i is a set of attributes that forms the basis of the i-th level of grouping (i starts from 1). $g_1 = \{NULL\}$.
4) O is a list for order specification, where o_i is for group level i. For $1 \leq i < |O|$, o_i takes a value of either "ASC" or "DESC", and ordering attributes are those in g_{i+1} but not in g_i. For $i = |O|$, o_i contains two ordered lists: *attributes* and *orders* ("ASC" or "DESC"). Elements in the same position of the two lists form an ordering, and *attributes* may contain any attribute not in G.

A spreadsheet presents only its base relation R, and multiple spreadsheets can present the same relation. Given R, it is straightforward to construct an initial spreadsheet by directly inheriting the columns, and leaving grouping and ordering to be empty.

Definition 2 (**Base Spreadsheet**)**.** For relation R, its *base spreadsheet* is $S^0(R, C^0, G^0, O^0)$, where C^0 is the set of columns in R, and G^0 and O^0 are both empty lists.

In this paper, tuples in R can be changed anytime, and the spreadsheet always retrieves the latest data. However, we require that the columns of the base relation R remain unchanged in the *life–time* of a spreadsheet S, which starts from the creation of S_0 to the destruction of the latest version of S. If R does change, we create a new base spreadsheet.

III. The Spreadsheet Algebra

In this section, we present operators on spreadsheets and define their semantics.

A. Basic Data Organization Operators

Ordering and grouping are crucial operators in a spreadsheet even though they do not change the actual "content". We begin by studying these data organization operators first.

Grouping (τ). The grouping operator τ takes as parameters *grouping-basis* (a set of attributes) and *order* ("DESC" or "ASC"). It groups tuples with equal values in all elements of the *grouping-basis* and order the groups in the *order* specified. A new level of grouping is created when and only when *grouping-basis* contains a superset of attributes of any existing grouping basis. Tuples in previously finest level of groups are further grouped according to attributes newly specified. Those new groups (not the tuples in each group) are ordered according to $order$. Denote as L the ordering attributes at the finest level before applying τ. Tuples inside each new group are grouped by a new ordered list o_L, which contains all elements that are in L but not in *grouping-basis*. We use the subtraction sign to denote this "list subtraction" operation: $o_L = L - grouping\text{-}basis$. Note that L is unchanged after the subtraction.

TABLE II
Car Database After Grouping by Condition

ID	Model	Price	Year	Mileage	Condition
872	Jetta	\$15,000	2005	50,000	Excellent
901	Jetta	\$16,000	2005	40,000	Excellent
304	Jetta	\$14,500	2005	76,000	Good
723	Jetta	\$17,500	2006	39,000	Excellent
725	Jetta	\$18,000	2006	30,000	Excellent
423	Jetta	\$17,000	2006	42,000	Good
132	Civic	\$13,500	2005	86,000	Good
879	Civic	\$15,000	2006	68,000	Good
322	Civic	\$16,000	2006	73,000	Good

We now give the formal definition for grouping. We assume we start from version j of a spreadsheet – $S^j(R^j, C^j, G^j, O^j)$. As in Section II, we use subscripted lowercase letter to denote elements in a set or list represented by its corresponding capital form (e.g., o_1^j means the first element in O^j). Following standard convention, we use $o_1^j.attributes$ and $o_1^j.orders$ to access the components of ordering specification o_1^j.

Definition 3 (**Grouping**)**.** $\tau_{grouping\text{-}basis,order}(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^j, C^j, G^{j+1}, O^{j+1})$, where

1) $g_i^{j+1} = g_i^j$ for $1 \leq i \leq |G^j|$, and $g_i^{j+1} = grouping\text{-}basis$ for $i = |G^j| + 1$;
2) $o_i^{j+1} = o_i^j$ for $1 \leq i \leq |O^j|$; for $i = |O^j| + 1$, $o_i^{j+1}.attributes = o_i^j.attributes - grouping\text{-}basis$, and $o_i^{j+1}.orders$ is the corresponding sub-list of $o_i^j.orders$.

Example 1. We start from spreadsheet S^j in Table I, where cars are grouped by Model (DESC) and then Year (ASC), and ordered in the finest groups by Price (ASC). After operation $\tau_{\{Year,Model,Condition\},ASC}(S^j)$, we create the fourth level of grouping with relative grouping basis of Condition. The result table is shown in Table II.

Ordering (λ). It takes as parameters (*attribute, order, l*) and orders tuples in the l-th level groups by $attribute$ (ASC or DESC, as specified in $order$). Denote the number of grouping levels as n. As explained in Sec. II, ordering attributes in group levels other than the n-th are dictated by the grouping. If an ordering attribute is specified in the i-th level groups ($i < n$), and it is different from the current ordering attribute imposed by grouping, all groupings from level-$(i + 1)$ and beyond are destroyed (and so are any computed values, such as aggregations, based on these groupings).

While destruction of groups creates no algebraic or semantic difficulty, it can be confusing for a user. As such, order specifications that destroy grouping is permitted in our implementation only if there are no aggregates present on the grouping that will be lost (the aggregates have to be projected out before such operations are allowed).

Definition 4 (**Ordering**)**.** $\lambda_{attribute,order,l}(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^j, C^j, G^j, O^{j+1})$, and

1) If $1 \leq l < |G^j|$ and $attribute \notin (g_{l+1}^j - g_l^j)$: for $i < l$, $g_i^{j+1} = g_i^j$, $o_i^{j+1} = o_i^j$; for $i = l$, $g_i^{j+1} = g_i^j$, $o_i^{j+1} =$

order; for $i > l$, $g_i^{j+1} = NULL$, $o_i^{j+1} = NULL$.

2) If $1 \leq l < |G^j|$ and $attribute \in (g_{l+1}^j - g_l^j)$: $G^{j+1} = G^j$, $o_l^{j+1} = order$, and $o_i^{j+1} = o_i^j$ for $i \neq l$.
3) If $l = |G^j|$, then $G^{j+1} = G^j$. And, if $\exists i$ such that *attribute* $\in g_i^j$, $O^{j+1} = O^j$. Let $|G^j| = k$. If $attribute \in o_k^j.attributes$, change corresponding entry in $o_k^j.attributes$ to *order* to obtain o_k^{j+1}; else, add *attribute* and *order* to the end of respective list in o_k^j to obtain o_k^{j+1}.

Example 2. We start from spreadsheet S^j in Table I as in Example 1. If we apply $\lambda_{Mileage,ASC,3}(S^j)$, we further order cars by Mileage in the finest level of groups. If we apply $\lambda_{Mileage,ASC,2}(S^j)$, in level-2 groups, we destroy the grouping at level 3 (relative grouping basis of Year).

B. Basic Data Manipulation Operators

A desirable property of a query language is relational completeness [6], [5], meaning it can express all queries expressible with relational algebra. Among all relational operators, selection (σ), projection (π), Cartesian product ($\times$), set union ($\cup$), and set difference (-) form a *complete set*, since any other operators can be expressed as a sequence of operators from this set [4] (theorem 2.1). For this reason, we adopt this *complete set* of relational operators as the foundation of our direct manipulation interface. We also use the same set of abbreviations as relational algebra. For their relational counterparts, we subscript the abbreviations with "*r*" (for example, $\times_r$ for relational product). Since operators are confined within the spreadsheet model, there are crucial differences between our operators and their relational counterparts, as we will describe shortly. Other important operators include join, aggregation, formula computation, and duplicate elimination. The interface design of all operators will be presented in Section VI. Before detailing the operators, however, we introduce two related concepts: *stored spreadsheet* and *computed column*.

The spreadsheet is designed such that it should be sufficient to present only one spreadsheet to the user at any time. Not having to manipulate multiple spreadsheets simultaneously makes the spreadsheet model more user-friendly. This does create a problem for binary operators like Cartesian product, where two spreadsheets are involved. This is where *stored spreadsheet* comes in. We allow a spreadsheet to be stored and later re-loaded, regardless of the number of operations it went through. Binary operations can now be performed between a stored spreadsheet and the *current spreadsheet* (the one currently presented to the user).

Another design consideration is to provide users the power of analytical processing in the spreadsheet. General spreadsheet applications like Microsoft Excel are essential to modern business, partly because users can define formulas on the spreadsheet and do calculations and analysis [7]. We define a *computed column* for the result column of computing a formula over the existing spreadsheet. The essential property of computed columns is automatic updates. Once a user has defined such a column, the user expects it to reflect the value correctly even as the database or spreadsheet is updated.

We now introduce formal definition for each operator.

Selection (σ). Let F be a condition that may involve:

1) Atomic predicates in the form of $A\ OP\ B$, where A and B can be column names or constants (but not both being constants), with optional arithmetic or string operators (for example, $2 \times a$ or $a + b$), and OP is any comparison operator (e.g., ">"),
2) Expressions built from connecting items in (1) with "AND", "OR", or "NOT".

Definition 5 (Selection). $\delta_F(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^{j+1}, C^j, G^j, O^j)$, where for all tuples t in R^j, $t \in R^{j+1}$ if and only if F applied to t is true.

Projection (π). Projection takes a single parameter, $column$, for the column to be removed from the spreadsheet.

Notice the differences between π and its relational counterpart π_r: 1) π only removes one column at a time, while π_r can remove multiple columns in one go; 2) Therefore it is more natural to specify as parameter to π the column to be removed, whereas the parameter to π_r is the set of columns to be retained. No duplicate elimination is performed after projection, since a spreadsheet is defined as a recursive *multi-set*. Grouping and ordering is retained through projection.

Definition 6 (Projection). $\pi_{column}(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^j, C^{j+1}, G^j, O^j)$, where $C^{j+1} = C^j - column$.

Cartesian Product ($\times$). It is possible to compute the Cartesian product of the current spreadsheet S^j with a stored spreadsheet $S_s^k(R_s^k, C_s^k, G_s^k, O_s^k)$ (we use subscript *s* to denote "stored spreadsheet"). S_s^k and S^j can present different base relations. To compute $S_s^k \times S^j$, we perform relational product on R^j and R_s^k and apply grouping and ordering of S^j on the result to maintain coherence of presentation. This means that product is not symmetric: $S^j \times S_s^k \neq S_s^k \times S^j$ even after reordering columns, since the grouping and ordering would be different. All computed columns are updated such that computation is based on the product.

Definition 7 (Cartesian Product). $S^j(R^j, C^j, G^j, O^j)) \times S_s^k(R_s^k, C_s^k, G_s^k, O_s^k)) = S^{j+1}(R^j \times_r R_s^k, C^j \cup_r C_s^k, G^j, O^j)$.

Set operators. Like product, set union ($\cup$) and set difference (-) operate on two spreadsheets each time. Grouping and ordering of the current spreadsheet S^j are used for the result. Note that union, like product, is asymmetric, because of this notion of current spreadsheet versus the second stored spreadsheet. The two spreadsheets must be compatible (having the same set of columns, excluding computed attributes). No duplicate elimination is performed. The union (resp. difference) is computed using standard multi-set semantics, so that the union of a tuple and its duplicate are two identical tuples, and the difference $\{t, t\} - \{t\}$ is $\{t\}$. Also, computed attributes do not participate in set operators. Instead, computed attribute columns in the base spreadsheet are retained, and recomputed based on the new set membership.

Definition 8 (Set Union). $S^j(R^j, C^j, G^j, O^j)) \cup S^k_s(R^k_s, C^k_s, G^k_s, O^k_s)) = S^{j+1}(R^j \cup_r R^k_s, C^j, G^j, O^j)$.

Definition 9 (Set Difference). $S^j(R^j, C^j, G^j, O^j)) -_r S^k_s(R^k_s, C^k_s, G^k_s, O^k_s)) = S^{j+1}(R^j -_r R^k_s, C^j, G^j, O^j)$.

Join ($\bowtie$). Assume $S^j(R^j, C^j, G^j, O^j)$ is the current spreadsheet, and $S^k_s(R^k_s, C^k_s, G^k_s, O^k_s)$ is a stored spreadsheet. We can join the two on any condition F that is supported by SQL. To compute the join, we perform relational join with F as the join condition on R^j and R^k_s and inherit grouping and ordering from S^j. The resultant spreadsheet is used as the current spreadsheet, and all computed columns are updated accordingly. Join can be emulated by product followed by selection.

Definition 10 (Join). $S^j(R^j, C^j, G^j, O^j)) \bowtie S^k_s(R^k_s, C^k_s, G^k_s, O^k_s)) = S^{j+1}(R^j \bowtie_{r,F} R^k_s, C^{j+1}, G^j, O^j)$, where

1) $\bowtie_{r,F}$ means a relational join with condition F,
2) C^{j+1} contains all computed columns, as well as all columns in $R^j \bowtie_{r,F} R^k_s$.

Aggregation (η). Any standard aggregation operator (*e.g., sum, avg*) can be computed on a selected column. The operator computes the aggregate over all attribute values in that column within a group, at any level. Thus η takes parameters f(function), c (column over which aggregation is computed), and l (level of groups). Recall that a spreadsheet is always at least grouped by NULL. So if grouping has not been applied, the aggregation is computed over all values in the column in the entire spreadsheet. In general, grouping may have been applied, recursively. In such a case, the aggregation is computed over a group at the specified level, not necessarily the innermost one. In particular, aggregates may still be computed over the entire spreadsheet, across all groups. Note that all operators apply to individual tuples and not to intermediate groups. Thus the result of COUNT is the number of tuples in the group being counted, and not the number of sub-groups in the group, even if sub-groups are present.

The next question is where to store and display the result of the aggregation. There is only one result value per aggregation group. We could store the aggregation results in a separate table and join this with the base table as needed. However, this can be confusing for the user and makes many subsequent queries hard to specify. Therefore, we choose to forgo normalization and store the result of aggregation in an additional computed column (which is automatically added by the aggregation operator) with the value in this column repeated for all rows in each aggregation group. Table III shows an example, where an extra column "Avg_Price" is computed for cars of the same Model and Year.

Definition 11 (Aggregation). $\eta_{f,c,l}(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^j, C^j \cup \{column\}, G^j, O^j)$, where $column$ is the aggregation result column computed from $f(c)$ at group level l.

Formula Computation (FC) (θ). This operator provides facility for users to perform mathematical operations on columns and create a computed column from the result. It takes parameter f (formula to compute). Each record in the result column is computed from an arithmetic expression involving values in the same row (in one or more columns). For example, from a sales table, a user wants a formula *revenue* for each product, computed as "*price* $*$ *quantity*". As a computed column, the result column is automatically updated when underlying data is changed. This new column can be used in the same way as any other columns in the database.

TABLE III
CAR DATABASE – AVERAGE PRICE BY MODEL AND YEAR

ID	Model	Price	Year	Mileage	Avg_Price
304	Jetta	$14,500	2005	76,000	$15,167
872	Jetta	$15,000	2005	50,000	$15,167
901	Jetta	$16,000	2005	40,000	$15,167
423	Jetta	$17,000	2006	42,000	$17,500
723	Jetta	$17,500	2006	39,000	$17,500
725	Jetta	$18,000	2006	30,000	$17,500
132	Civic	$13,500	2005	86,000	$13,500
879	Civic	$15,000	2006	68,000	$15,500
322	Civic	$16,000	2006	73,000	$15,500

Definition 12 (FC). $\theta_f(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^j, C^j \cup \{column\}, G^j, O^j)$, where $column$ is the result column computed from R^j based on the current grouping.

Duplicate Elimination (DE) (δ). Since the spreadsheet model operates on multi-sets, duplicates are allowed in projection, grouping, and set union/difference. This is similar to relational implementations, but different from pure relational algebra. Where duplicate elimination is required, it must be invoked explicitly through the DE operator, which removes all duplicates from the current spreadsheet, similar to "distinct" in SQL. As a result of DE, computed columns (including both aggregation and FC) need to be re-computed. Ordering and grouping are not affected, since duplicates are already placed together where it matters.

Definition 13 (DE). $\delta(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^{j+1}, C^j, G^j, O^j)$, where $\forall t \in R^j$, $t \in R^{j+1}$, and $\forall t_1, t_2 \in R^{j+1}$, $t_1 \neq t_2$.

C. Additional Housekeeping Operators

As introduced in Section III-B, we can save the current spreadsheet anytime, and should also be able to load a saved spreadsheet, either for reading or for operations with the current spreadsheet. Thus we have **Save**, **Open**, and **Close** operators. In addition, we supply the **Renaming** operator for changing the name of a column.

IV. PROPERTIES OF THE SPREADSHEET ALGEBRA

The spreadsheet algebra is for building an expressive and usable direct manipulation interface. In the first subsection below we show that the algebra can emulate *core single-block SQL queries*.

While complex expressions can be developed in the spreadsheet algebra, it is important to remember that the purpose of this algebra is to enable the manipulation of a spreadsheet visible to the user. Operator precedence and commutativity play an important role in this regard, as will become clear later. In the second subsection below we deepen our understanding of these two vital properties of spreadsheet algebra operators.

A. Expressive Power

We define a *core SQL single-block query expression* to be a statement of the form:

SELECT $<$ projection-list $>$ $<$ aggregation-list $>$
FROM $<$ relation-list $>$
WHERE $<$ selection-predicate $>$
GROUP BY $<$ grouping-list $>$
HAVING $<$ group-selection-predicate $>$
ORDER BY $<$ ordering-list $>$

with the projection-list being a subset of the grouping-list and the ordering-list a subset of the projection-list union aggregation-list, where all lists are comprised of column names.

Theorem 1. *For every core SQL single-block query expression there exists an equivalent expression in the spreadsheet algebra such that the result of evaluating either expression against any set of relations is identical.*

Proof:

We prove the above theorem by providing a procedure for specifying a core SQL single-block query expression (denote it as *s*) with our algebra.

Step 1: One at a time, obtain the Cartesian product of each relation named in the relation-list, to obtain a single product working relation.

Step 2: Remove all join conditions from the where clause, if any; specify the remaining where clause using the selection operator.

Step 3. We specify each item in the grouping-list from left to right, using the grouping operator. The grouping operator takes as input a grouping column, and the level of this new grouping. We create a new level of grouping with each item.

Step 4. Specify aggregations, which could appear in both the SELECT and ORDER BY clause. In SQL, when there are multiple grouping levels (multiple items in the group-list), aggregation is computed over the finest level. We specify each aggregation using the aggregation operator accordingly.

Step 5. Specify the HAVING clause. Since we already created aggregations columns for the HAVING clause in step 4, we can apply selection operator on aggregation columns as required in the given group-selection-predicate.

Step 6. Specify ORDER BY clause using the ordering operator. As in step 4, the ordering is specified over the finest level of grouping.

Step 7: Project out all columns not included in the projection-list, one at a time.

We have thus completed the specification of a core single-block SQL query.

B. Commutativity

If operator $\aleph_i$ commutes with $\aleph_j$, the order of evaluation of these two operators does not affect the result. Commutativity among the *complete set* of relational operators has been studied in [8]. In relational algebra, under the condition that attributes in selection predicates are retained in projection, selection commutes with all operators. Under the same condition, projection commutes with every operator except set difference [8]. In spreadsheet algebra, all binary operators (set union, set difference, join, and product) involve a stored spreadsheet, and this impacts the applicability of commutativity laws for reasons that follow. When commuting selection with a set difference, we use the following formula [8]:

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2) \qquad (1)$$

As we can see, this requires selection to be applied to a stored spreadsheet first. Since the other spreadsheet already occupies the data view, this can only be done in the background. This is against our direct manipulation principle, which dictates all changes to be seen directly by the user. A *point of non-commutativity* is created whenever a binary operator (set union, difference, join, product) is applied, meaning any operator instance after this point does not commute with instances before that. As explained above, since these binary operators manipulate both the current spreadsheet and a stored spreadsheet, distribution is not possible in the spreadsheet model. There is no way to apply operators to a spreadsheet being read in prior to its being read in for, say, a set union.

We say that a spreadsheet operator instance p *precedes* operator instance q if q requires columns created by p or q removes a column that p requires. For example, column-creating operators (e.g., aggregation) precedes selection that uses the aggregation result column. In order for two operator instances to commute, neither of them can precede the other.

For commutativity in spreadsheet algebra, we have the following observation:

Theorem 2. *In the spreadsheet algebra, selection, projection, FC, DE, and aggregation commute with one another, with themselves, and with grouping and ordering, provided that all precedence relations are satisfied.*

Proof sketch: We first verify the pair-wise commutativity for the five unary data manipulation operators – selection, projection, aggregation, formula computation (FC), and duplicate elimination (DE). We observed that they all commute with each other in spreadsheet algebra. Some pairs appear surprising, for example, aggregation and selection, which do not commute in relational algebra. They commute in spreadsheet algebra for two reasons: 1) aggregation result is stored as a separate column with repeated values instead of a single value, and 2) result values are updated once underlying data is changed (for example, by selection). A similar argument applies to DE and aggregation. We next examine whether the five operators commute with grouping and ordering operator. Obviously grouping and ordering do not commute with each other (for example, some ordering can destroy grouping), but they commute with data manipulation operators. The general reason for this is that grouping and ordering are maintained by data manipulation operators.

V. Query Modification

In an interactive query environment, users often find it useful to modify their query to obtain the desired query results. Frequently, these modifications are small, such as changing a threshold parameter in a selection condition. Suppose the user has performed *n* operations $\{\aleph_1, \aleph_2, ...\aleph_n\}$ in sequence (that is, $\aleph_i$ is executed earlier than $\aleph_j$ when $i < j$) on the data, and she intends to modify a condition specified in the *i*-th operation. In the naive case, the user has to begin from scratch, and repeat all *n* operations, making the desired change in the *i*-th operation. If the interface provides an UNDO facility, the user may not have to start over from scratch. But she still has to back up to the *i*-th operation, and re-specify everything from there onwards.

What we would like instead is for the user to be able to specify a change only to the one affected *i*-th operation, and have the system take care of the rest. To be able to accomplish this, we need a notion of *query state*. The system keeps track of the history of operators specified by the user. The system could undo all operations back to the *i*-th and then re-do from there again. However, this is likely to take too long. The commutativity property of spreadsheet algebra, as stated in Theorem 2, can be used to reduce this cost substantially.

A. Query State

Instead of keeping every user action from the beginning, we keep query history until only as far back as we expect to be able to permit rewriting efficiently, which is the most recent point of non-commutativity. From a user perspective what this will lead to is that queries specified on a single sheet can be changed as needed, but where data from other sheets has been pulled in we cannot go back beyond.

For each selection or FC, we associate the predicate applied with the column(s) referenced in the predicate. With each column, we store the associated selection/FC predicates. This includes columns created by aggregation and FC.

For each projection, we retain a list of columns projected out.

For each aggregation, we retain, associated with the corresponding aggregate column, a definition of the aggregate function applied, and the grouping it is applied to.

For each grouping and ordering, we retain the corresponding grouping-list or ordering-list.

Notice that we did not store the query state as an ordered list of manipulations, but rather as individual operators associated with objects they affected. On account of operator commutativity, we can generate a history that is equivalent to the actual history of the spreadsheet.

Theorem 3. *In a direct manipulation interface, modifying an operation in a sequence of operations without point of non-commutativity through query state change is the same as rewriting query history.*

Proof sketch: The proof of this theorem follows commutativity properties we established in Sec. IV-B. We store all operations, without ordering, in association with either the related columns or the operations themselves, so we are able to list all previous operations without order. Since operations in a sequence commute, in the absence of any point of non-commutativity, the order in which they are applied is immaterial. Hence changing any one operation from the query state has the same effect as changing it in the complete ordered list of operations.

B. Query Specification

Having introduced query state, we now show how to modify a previously specified query through query state. We could show the user the entire query state and let them specify what they wish to modify. However, non-technical users may have difficulty understanding the query state. Moreover, manipulating query state goes against the notion of direct manipulation of data, which is our objective. Instead, we selectively present history to the user as she attempts to redefine the invocation parameters of an operator, in a manner that we make precise next.

When a user begins to specify a selection predicate on a column, the user is given a list of selection predicates currently applied to that column, from the query state. The user then has an option of replacing a previously applied predicate with the one now being specified, or even of deleting the previously applied predicate altogether, without specifying a new predicate at all. History is rewritten, with the previously applied predicate removed, and replaced with the new one, if one is specified. Of course, the user also has the option of simply specifying the new predicate in addition to those previously specified. In this case, the new predicate is added "now", without rewriting history.

We can remove an existing selection predicate on any column, just as we can add a new one to any column. Putting these two together, we can also modify the selection predicate on any column.

We use an "inverse" projection operator to "reinstate" a column that has been projected out. We write this as $\Pi_{\bar{l}}(R)$. Since a column is either included or excluded in a projection, there is really no additional history to show. The semantics of the reinstatement are to rewrite history, and make it as if the projection never took place.

We can remove an aggregate column, provided that no operator depends on it. If a column that serves dependencies needs to be removed, all dependent columns must be removed first. Of course, we can always add a new aggregate column.

We can modify an existing grouping or ordering, provided that there is no operator that depends on it. Otherwise, those that depend on the grouping or the ordering should be removed first.

We now turn to the used car example. Suppose Sam initially started by searching for cars in year 2005, model "Jetta",

TABLE IV
RESULTS BEFORE QUERY MODIFICATION

ID	Model	Price	Year	Mileage	Condition
872	Jetta	$15,000	2005	50,000	Excellent
901	Jetta	$16,000	2005	40,000	Excellent
304	Jetta	$14,500	2005	76,000	Good

TABLE V
RESULTS AFTER QUERY MODIFICATION

ID	Model	Price	Year	Mileage	Condition
723	Jetta	$17,500	2006	39,000	Excellent
725	Jetta	$18,000	2006	30,000	Excellent
423	Jetta	$17,000	2006	42,000	Good

and mileage lower than 80k. Results should be grouped by condition and ordered in ascending order of price. After seeing the results (as shown in Table IV), he discovers that his budget allows him to purchase a newer car. Sam can now simply choose the "Year" column, and change previous condition of "Year = 2005" to "Year = 2006". If he prefers, he can also modify the mileage condition in a similar fashion. All results are updated to meet the new condition(s), and the specification of model, grouping and ordering remains effective, as shown in Table V.

VI. INTERFACE DESIGN

The reason to develop the spreadsheet algebra described above is to implement an effective and easy-to-use spreadsheet interface to a database system. We built a prototype named *SheetMusiq* to validate the ideas we presented, as part of the **Musiq** (**M**odel-driven **U**sable **S**ystem for **I**nformation **Q**uerying) effort at the University of Michigan [9].

SheetMusiq reflects all three principles of direct manipulation. First, users specify queries in SheetMusiq by mouse-clicks, with minimal keyboard input (e.g., for inputting constants to compare with). Most query operations are accessible with a *contextual* menu, which pops up when the user right-clicks a cell or column-header. It is *contextual* because it shows only options that are available for the current cell value type under current grouping and ordering. Second, SheetMusiq provides immediate and intuitive result presentation for users to easily specify conceptually difficult queries. It continuously presents the resultant spreadsheet after each manipulation, helping users to better adjust the next query step. Third, all user actions are reversible. Users can access query history (all historical manipulations) through a "History" menu, and the complete list of operations is shown as a numbered list, each with meaningful names. Users can do one-step or multi-step undo/redo of data manipulation. They can also do query modification to modify an operation in a sequence of operations, without having to repeat the effort to re-specify unchanged operations.

A. Design of Operators

We first introduce user interface design for each operator, then we re-visit the motivating problem in Section I-A using related operators.

Grouping. Grouping is accessed through a context menu. If the spreadsheet is already grouped by other column(s), the user is asked whether to add to the existing grouping (as the inner most level of grouping) or destroy the current grouping and use this new one instead. However, if there are aggregation columns that depend on the current grouping, user clicking on the latter option will trigger a reminder to first remove columns that depend on the grouping.

Ordering. Clicking a column header sorts the table according to the column in ascending order, and another click changes the sorting to descending order. The header of such a column has an up/down arrow, to show ascending/descending order. In the presence of grouping, the user is asked explicitly for the level of grouping to which the order should be applied. If the new ordering can destroy some grouping, the user is asked for confirmation to do so. If there are aggregates that depend on that grouping, this operation is not allowed, and user is suggested to project out the aggregates, if necessary.

Selection. Selection is accessed with a right-click of the mouse. If the click is on a cell instead of a column header, the user can choose to filter the results based on current cell value with another click, and the result is immediately shown.

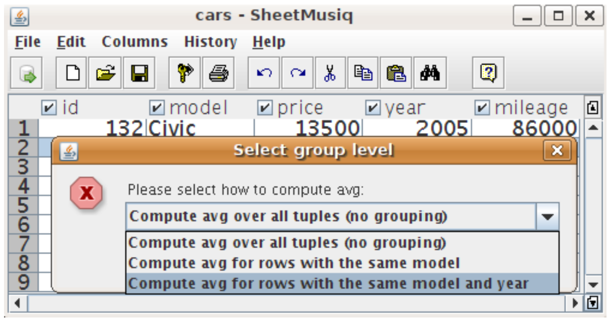
Projection. We present a checkbox to the left of each column header. By default all checkboxes are checked. Users can remove a column conveniently by unchecking the checkbox (Figure 1). Columns that are projected out can be restored from a drop down menu.

Cartesian Product and **Set operators**. These binary operators involve an additional stored spreadsheet (which is saved previously by a clicking on the "Save" button). Once an operator is called through a contextual menu, the user is presented with all stored-relations listed in a pop-up menu. The result is then presented as the current spreadsheet.

Join. Join also involves a stored spreadsheet. In addition to choosing a spreadsheet to join with, the user is prompted to graphically choose join conditions. Validity of join condition is checked and any invalid condition is reported to the user immediately. The result spreadsheet is shown in the screen as the current one.

Aggregation. A user can perform an aggregation function on an attribute by right clicking a cell and choosing "aggregation". She is then given a choice of aggregate function, and possibly the option to specify the grouping level on which the aggregation should be computed (when data is grouped by some column). Result column appears next to rightmost column.

Formula Computation. A dialog is shown, allowing the user to choose related columns and mathematical operators. The user can optionally give a name for the result column. Otherwise, the system automatically generates a name for it and reminds the user of the new column. The new column is added to the right of all existing columns.



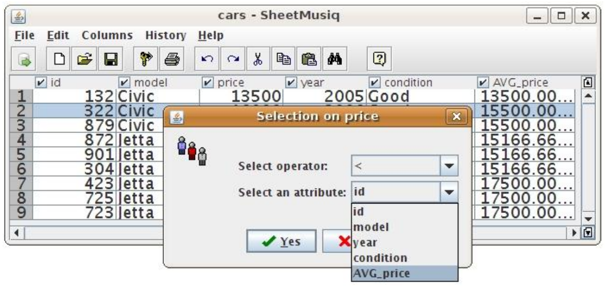
	id	model	price	year	mileage
1	132	Civic	13500	2005	86000

Please select how to compute avg:

Compute avg over all tuples (no grouping)

Compute avg over all tuples (no grouping)
Compute avg for rows with the same model
Compute avg for rows with the same model and year

Fig. 1. Aggregation under Grouping



	id	model	price	year	condition	AVG_price
1	132	Civic	13500	2005	Good	13500.00...
2	322	Civic				15500.00...
3	879	Civic				15500.00...
4	872	Jetta				15166.66...
5	901	Jetta				15166.66...
6	304	Jetta				15166.66...
7	423	Jetta				17500.00...
8	725	Jetta				17500.00...
9	723	Jetta				17500.00...

Select operator: <
Select an attribute: id

id
model
year
condition
AVG_price

Fig. 2. Compare Price with Avg_Price

Duplicate Elimination. DE is accessible by a single right-click, and all duplicates are removed.

Having introduced all operators, we show how Sam could use the operators to explore the car database. Recall that Sam likes cars of Year 2005 or later in good or excellent condition, and the rest is open-ended. Since he cares about Model and Price the most, he first groups the data by "Model" and "Year", and selects "Good" and "Excellent" condition. He also selects his Models of interest: "Jetta" and "Civic". Now he wants to know the average price for the Model and Year so that he does not overpay. To accomplish that, he computes the average price using Aggregation, where he is asked to compute average over all the cars or just cars of the same Model and Year (show in Figure 1). Choosing the latter leads the spreadsheet in Table III. Now he can filter out all cars more expensive than the average, as shown in Figure 2, where he chooses to compare "Price" with "AVG_price".

VII. Evaluation

In this section we experimentally evaluate the algebra with our prototype, SheetMusiq. The database server was PostgreSQL 8.3 (beta 2).

A. User Study

We now measure the usability of SheetMusiq with user studies. Since the interface is built specifically to help people without knowledge of any database query language (which rules out any direct SQL query interface), we want to compare SheetMusiq with a tool that meets the same requirement and with similar expressive power. Graphical query builders are the closest existing tools that satisfy the requirement. We found an abundance of such packages, and they are mostly similar. We chose Navicat for PostgreSQL [10] as a representative to compare with. All experiments were performed on a laptop (Intel Core 2 Duo 2.16GHz CPU, 2GB of RAM, and running Windows XP).

1) Methods: We recruited ten volunteers with no background in database query languages. Their ages ranged from 24 to 30, and they all have at least a bachelor's degree (in various fields).

The data and queries are from the TPC-H benchmark [11]. We used the demonstration dataset in the benchmark, which was 31MB in size. Since TPC-H queries are quite complex, some of them contain features that SheetMusiq does not yet support. Specifically, SheetMusiq does not support nested queries and queries with keyword "exist" and "case". This leaves us 10 queries out of the original 22 in the benchmark. In addition, we predefined views for queries involving many joins so that users always query a single table.

A brief tutorial (introduction with examples for both SheetMusiq and Navicat) was given to each subject prior to the study. At the end of the tutorial, subjects completed a sample query, with help available upon request from the examiner. Each participant then completed all queries in the query set, using both SheetMusiq and Navicat, separately. For each user, we gave her/him time to understand the query definition. We started measuring time when the user decided the query was fully understood and he/she was ready to specify the query. Since the software that is used first has a potential disadvantage, we alternate the order of which software was used first for the queries. In the end, each package was used first half the time.

For each subject, we measure speed (time for each task) and correctness. During the experiment, if a user did not finish the query in 900 seconds, the task was considered finished with wrong results, and the time was counted as 900 seconds.

2) Speed Results: We measure the time taken by the subjects for completing each query. Figure 3 shows the average time for each user to complete the 10 queries, using Navicat and SheetMusiq, respectively. Most queries were completed significantly faster with SheetMusiq than Navicat. Using the Mann-Whiteney test we found the speed result is statistically significant (with p-value < 0.002) for all queries except query 5, 7, and 10. For those three queries, the speed performance is comparable with both packages. We think the reason is that the three query tasks are relatively simple, and subjects can finish both in a short time. Furthermore, as shown in Figure 4, the standard deviation for SheetMusiq is much smaller on most queries, demonstrating the consistency of superior efficiency using SheetMusiq.

3) Correctness Results: We consider correctness of queries the subjects finished. Figure 5 shows the number of users that completed the queries correctly using the two approaches.

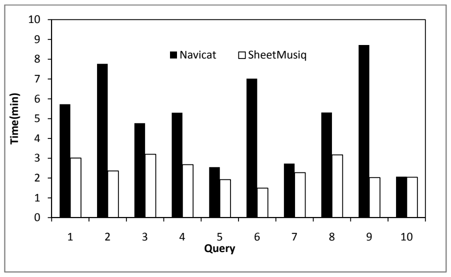


Fig. 3. Speed Result

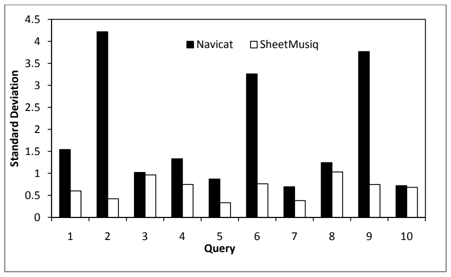


Fig. 4. Standard Deviation of Speeds

SheetMusiq users correctly finished more queries (95 out of 100) than Navicat users (81 out of 100). If we consider each query alone, we can not establish a statistically significant conclusion. Hence we consider the total number of correctly answered queries. Using Fisher's exact test we conclude that SheetMusiq is statistically better than Navicat (in leading to more correctly answered queries), with p value < 0.004.

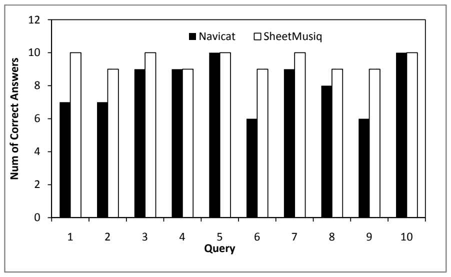


Fig. 5. Correctness Result

4) Analysis: We now analyze why better speed and higher accuracy were observed for SheetMusiq.

Navicat, like most other graphical query builders, has two separate windows for building a query – a graphical window where users manipulate with mouse-clicks and a text window for SQL query expression. Usually, only queries with simple selection, sorting, and joins can be built graphically, while the vast majority of the queries need to be completed by adding to the SQL query. SheetMusiq never reveals or requires the user to know a SQL query. This leads to several benefits, which directly led to the superior performance of SheetMusiq. First, we found that most users picked up SheetMusiq much faster than Navicat (also shown by results of the first two queries). Second, users never stuck on syntactical errors in SheetMusiq, which often happen in Navicat.

Many users had difficulties accomplishing certain tasks in Navicat but did that effortlessly in SheetMusiq, largely because Navicat (and almost all graphical query builders we surveyed) does not have direct manipulation support for database concepts other than simple selection, sorting, and join and SheetMusiq does. We now give a few examples of such tasks. First, selection based on aggregation. In Navicat, users have to resort to a *sub-query*, which is a very difficult concept for non-expert users. In SheetMusiq, this can be accomplished by an aggregation followed by selection, both via mouse-clicks. Second, grouping is much easier in SheetMusiq. In Navicat, users have no choice but to understand the concept and syntax of grouping, as well as many related restrictions (e.g., aggregation). This is very challenging for almost every participant. In SheetMusiq, users do not need to know the syntax (mouse right-click is sufficient), and they understand the concept faster with immediate visual feedback. Third, group-qualification. While many users struggled with the "having" clause in Navicat, they found it very intuitive to filter groups with mouse-clicks. This list could go much longer if space were permitted.

TABLE VI
SUBJECTIVE RESULTS

Question	Answers	Count
Which package do you prefer to use?	Navicat SheetMusiq	0 10
Seeing data helps formulate queries	Yes No	10 0
Progressive refinement is better than specifying a query all at once	Yes No	8 2
Database concepts are easier in SheetMusiq	Yes No	10 0

5) Subjective Results: Besides numerical evidence, each participant was asked for opinions on which program they would prefer and whether certain features are desirable. Results are presented in Table VI. All subjects preferred SheetMusiq over Navicat, and all agreed that being able to see the data helps formulate a query (second question). They also found that many concepts (e.g., group-qualification) are more intuitive and easier to understand in SheetMusiq (fourth question). Eight of ten subjects preferred progressive refinement of a query over specifying it all at once (third question).

VIII. RELATED WORK

Direct manipulation [1], although a crucial concept in the user interface field, is seldom mentioned in database literature. Pasta-3 [12] is one of the earliest efforts attempting a direct manipulation interface for databases, but its support of direct manipulation is limited to allowing users to manipulate *a query* *expression* with clicks and drags. Tioga-2 [13] (later developed into DataSplash [14]) is a direct manipulation database visualization tool, and its visual query language allows specification with a drag-and-drop interface. Its emphasis, however, is on visualization instead of querying.

Spreadsheets have proven to be one of the most user-friendly and popular interfaces for handling data, partially evidenced by the ubiquity of Microsoft Excel. FOCUS [15] provides an interface for manipulating local tables. Its query operations are quite simple (e.g., allowing only one level of grouping and being highly restrictive on the form of query conditions). Tableau [16], which is built on VizQL [17], specializes in interactive data visualization and is limited in querying capability. Spreadsheets have also been used for data cleaning [18], logic programming [19], visualization exploration [20], and photo management [21]. Witkowski et al [22] proposed SQL extensions supporting spreadsheet-like computations in RDBMS.

In addition to desktop tools, we have seen many online database query and management tools using spreadsheet interfaces. For example, Zoho DB [23] allows importing, creation, querying, and visualizing databases online. Zoho DB's querying capability, however, is still primitive (allowing only simple filtering and sorting, with the rest resorting to SQL). Dabble DB [24] is similar to Zoho DB, with the additional feature of supporting grouping.

Much effort has been spent on a visual querying interface for relational databases, starting with Query-by-Example [25]. Query-by-Diagram [26] allows users to query by manipulating ER-diagrams. T. Catarci et al. surveyed many early interfaces and visual query languages in [27]. [28] uses ontologies to help users with their vocabulary in formulating queries. VisTrails [29] provides a visual interface for users to keep track of incrementally specified workflows and to modify them incrementally. [30] proposes a visual query language that enables users to query a database by manipulating "schema trees".

IX. Conclusion

A spreadsheet-like "direct manipulation" query interface is desirable for non-technical database users but challenging to build. In this paper, we design a spreadsheet algebra that enables the design of an interface that: i) continuously presents the data to users, after each data manipulation, ii) divides query specification into progressive refinement steps and uses intermediate results to help users formulate the query, iii) provides incremental reversible data manipulation actions, iv) enables the user to modify an operation specified many steps earlier without redoing the steps afterwards, and v) allows the user to specify at least all single-block SQL queries while shielding her from complex database concepts. We built a prototype, SheetMusiq, with our algebra and evaluated it using user studies with non-technical subjects, in comparison with a commercial graphical query builder. Results show that the direct manipulation interface leads to easier and more accurate specification of queries, and it is welcomed by non-technical users.

X. Acknowledgments

This work was supported by the US National Science Foundation under grant IIS-0438909 and IIS-0741620.

References

[1] B. Shneiderman, "The future of interactive systems and the emergence of direct manipulation," *Behaviour & Information Technology*, vol. 1, no. 3, pp. 237–256, 1982.

[2] ——, "Direct manipulation: a step beyond programming languages," *IEEE Computer*, vol. 16, no. 8, pp. 57–69, 1983.

[3] ——, "A computer graphics system for polynomials," *The Mathematics Teacher*, vol. 67, no. 2, pp. 111–113, 1974.

[4] J. Ullman, *Principles of database and knowledge-base systems, Vol. 1*. Computer Science Press, Inc. New York, NY, USA, 1988.

[5] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw-Hill Boston, 2003.

[6] A. V. Aho and J. D. Ullman, "Universality of data retrieval languages," in *ACM POPL*, 1979.

[7] A. Witkowski, S. Bellamkonda, T. Bozkaya, A. Naimat, L. Sheng, S. Subramanian, and A. Waingold, "Query by excel," in *VLDB*, 2005, pp. 1204–1215.

[8] J. Ullman, *Principles of database and knowledge-base systems, Vol. 2*. Computer Science Press, Inc. New York, NY, USA, 1988.

[9] "Database usability research at university of michigan," http://www.eecs.umich.edu/db/usable/.

[10] "Navicat," http://pgsql.navicat.com/.

[11] "Transaction processing performance council." TPC-H Benchmark Specification, Version 2.6.1.

[12] M. Kuntz and R. Melchert, "Pasta-3's graphical query language: Direct manipulation, cooperative queries, full expressive power," in *VLDB*, 1989, pp. 97–105.

[13] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff, "Tioga-2: A direct manipulation database visualization environment," in *ICDE*, 1996, pp. 208–217.

[14] C. Olston, A. Woodruff, A. Aiken, M. Chu, V. Ercegovac, M. Lin, M. Spalding, and M. Stonebraker, "Datasplash," in *SIGMOD*, 1998, pp. 550–552.

[15] M. Spenke, C. Beilken, and T. Berlage, "Focus: The interactive table for product comparison and selection," in *UIST*, 1996, pp. 41–50.

[16] "Tableau software," http://www.tableausoftware.com/.

[17] P. Hanrahan, "Vizql: a language for query, analysis and visualization," in *SIGMOD*, 2006, p. 721.

[18] V. Raman and J. M. Hellerstein, "Potter's wheel: An interactive data cleaning system," in *VLDB*, 2001, pp. 381–390.

[19] M. Spenke and C. Beilken, "A spreadsheet interface for logic programming," in *CHI*, 1989, pp. 75–80.

[20] T. J. Jankun-Kelly and K.-L. Ma, "A spreadsheet interface for visualization exploration," in *IEEE Visualization*, 2000, pp. 69–76.

[21] S. Kandel, A. Paepcke, M. Theobald, and H. Garcia-Molina, "The photospread query language," Stanford Univ., Tech. Rep., 2007.

[22] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian, "Spreadsheets in rdbms for olap," in *SIGMOD*, 2003.

[23] "Zoho db & reports," http://db.zoho.com/.

[24] "Dabble db - online database," http://dabbledb.com/.

[25] M. M. Zloof, "Query-by-example: the invocation and definition of tables and forms," in *VLDB*, 1975, pp. 1–24.

[26] T. Catarci and G. Santucci, "Query by diagram: A graphical environment for querying databases," in *SIGMOD*, 1994, p. 515.

[27] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini, "Visual query systems for databases: A survey," *J. Vis. Lang. Comput.*, vol. 8, no. 2, pp. 215–260, 1997.

[28] T. Catarci, P. Dongilli, T. D. Mascio, E. Franconi, G. Santucci, and S. Tessaris, "An ontology based visual tool for query formulation support," in *ECAI*, 2004, pp. 308–312.

[29] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. Silva, "Querying and re-using workflows with vistrails," in *ACM SIGMOD*, 2008.

[30] C. Koch, "A visual query language for complex-value databases," *ArXiv Computer Science e-prints*, 2006.