PACKAGEBUILDER: From Tuples to Packages

Matteo Brucato* Rahul Ramakrishna* Alexandra Meliou* Azza Abouzied§

*School of Computer Science
University of Massachusetts
Amherst, USA
{matteo,rahulram,ameli}@cs.umass.edu

§Computer Science
New York University
Abu Dhabi, UAE
azza@nyu.edu

ABSTRACT

In this paper, we present PACKAGEBUILDER, a system that extends query engines to support package generation. A package is a collection of tuples with certain global properties defined on the collection as a whole. In contrast to traditional query answers, where each answer tuple needs to satisfy the query predicate constraints, each answer package needs to satisfy global constraints on the collection of tuples: e.g., a package of recipes that collectively do not exceed 2,200 calories. PACKAGEBUILDER introduces simple extensions to the SQL language to support package-level predicates, and includes a simple interface that allows users to load datasets and interactively specify package queries. Our system allows users to interactively navigate through the result packages, and to provide feedback by fixing tuples within a package. PACKAGEBUILDER automatically processes this feedback to refine the package queries, and generate new sets of results.

1. INTRODUCTION

Traditional database queries define constraints (selection predicates) that each tuple in the result needs to satisfy. While traditional SQL queries are undoubtedly expressive and powerful, they fall short in scenarios that require a set of answer tuples to satisfy constraints collectively. Such scenarios arise in a variety of applications:

Investment portfolio: A broker wants to construct an investment portfolio for one of her clients. The client has a budget of $50K, wants to invest at least 30% of the assets in technology, and wants a balance of short-term and long-term options. The broker cannot select each stock option individually, but rather needs to find a stock package that satisfies all these constraints collectively.

Meal planner: An athlete needs to put together a dietary plan in preparation for a race. She wants a high-protein set of three meals for the day, that are between 2000 and 3000 calories in total. All meals should be gluten-free. It is easy to exclude meals that include gluten, as this condition can be checked for each meal (tuple) individually with a regular selection predicate. Other constraints need to be verified collectively over the entire package.

Vacation planner: A couple wants to organize a relaxing vacation at a tropical destination. They do not want to spend more than $2,000 in flights and hotel combined, and they want to be in walking distance from the beach, unless they can fit a rental car in their budget, in which case they are willing to stay farther away. Building the ideal vacation package is challenging, as the choice of hotel affects the choice of other elements in the package (e.g., flights and car rental).

In this paper we present PACKAGEBUILDER, a system that augments database functionality to support the creation of *packages*. A *package* is a collection of tuples that individually satisfy *base constraints* and collectively satisfy *global constraints*. The base constraints are equivalent to regular selection predicates, and can be evaluated individually for each tuple. For example, in the meal planner application, the gluten-free restriction is a base constraint, as it can be verified independently on each meal. In contrast, the requirement that total calories should be within 2,000 to 2,500 calories: it cannot be evaluated on each meal individually, but needs to be assessed over a collection of meals.

In this paper we show how to build such packages from database data using *package builder queries* (PBQs). Our system addresses three main challenges:

Language specification: Even though many use cases motivate support for PBQs, this class of queries remains largely unsupported with few tools targeting domain-specific packages (e.g., CourseRank supports building course packages [2]). As part of this work, we will present PaQL, a declarative query language that supports package specifications. PaQL is designed with simple extensions to standard SQL, so those familiar with SQL should find it intuitive and easy to use (Section 3).

Interactive specification: Even traditional SQL queries can often be challenging for novice DBMS users to specify. To enable user-friendly database applications, several systems now employ application-independent visual metaphors for SQL query specification [9, 3, 4]. PBQs are fundamentally harder to express and evaluate compared to traditional SQL, therefore, it is increasingly important to provide visual paradigms to guide users through building a query, as well as navigating and possibly refining the results. PACKAGEBUILDER offers an interactive representation of datasets that guides users in specifying base and global constraints for their packages. The system interface also allows users to easily navigate through the solution space by visualizing the result space, and to refine the result packages. (Section 4).

Evaluation: In traditional database queries, the size of the answer is polynomial in the size of the input data. This is not true for package queries: If n tuples satisfy the base constraints of a package, there are $\Omega(2^n)$ candidate packages that can satisfy the user's global constraints. This makes the evaluation of PBQs particularly

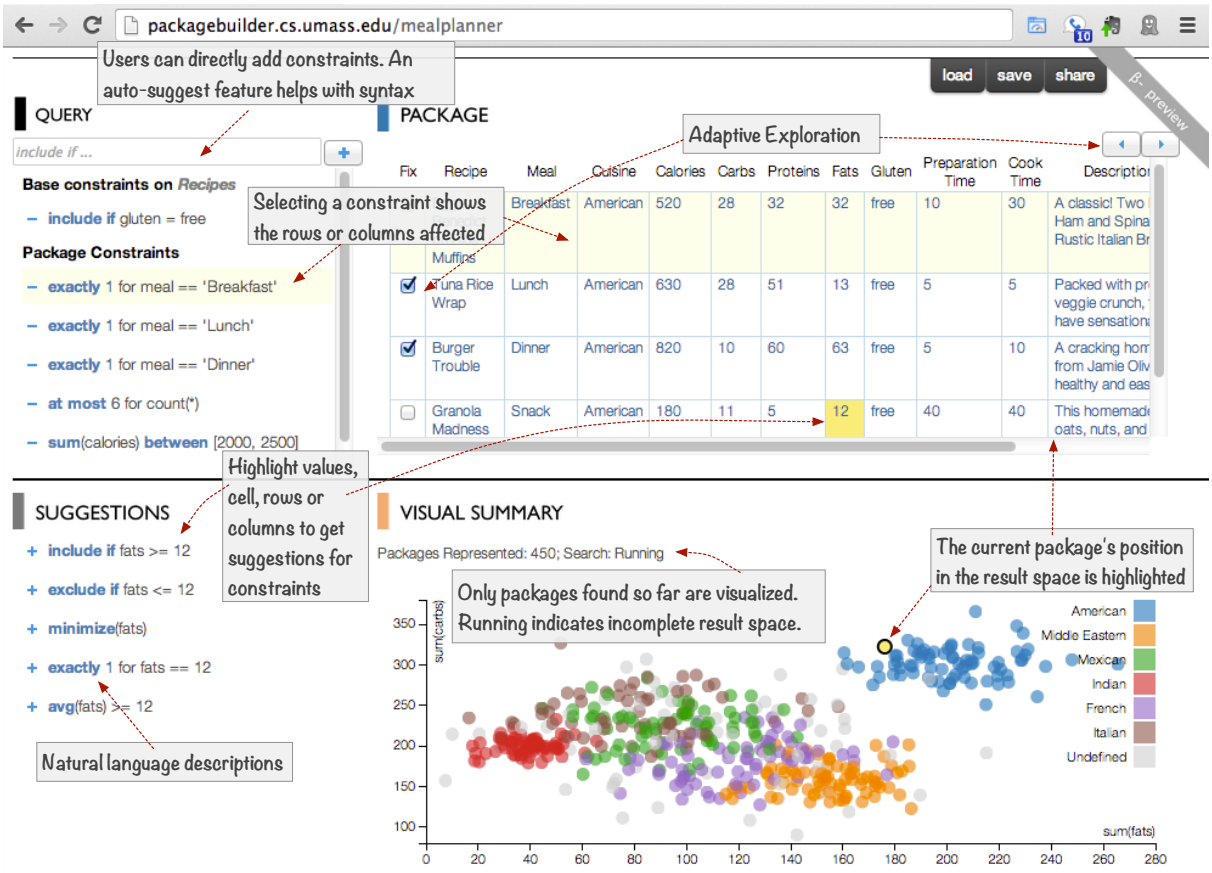

Figure 1: PackageBuilder's visual interface

challenging. With an exponential search space, efficiently searching for packages that satisfy our users' constraints requires applying non-trivial pruning techniques and search heuristics (Section 5).

We proceed to describe the main three aspects of our system that are motivated by these challenges.

2. RELATED WORK

Package queries are instances of *constraint satisfaction problems* (CSP) [6], a well-known class of NP-complete problems. As we illustrate in Section5, package queries can often be encoded as linear programs [5]. This facilitates the use of fast, off-the-shelf LP solvers. Package queries can be used to provide *set-based recommendations* such as those provided by CourseRank [2]. PaQL is powerful and can easily express pre-requisite constraints typical of course package recommendation systems.

Package queries are also related to *how-to queries* [8], which compute database instances that satisfy a set of base and global constraints. Unlike package queries, how-to queries can create new base tuples to satisfy constraints. How-to queries, however, cannot produce multisets.

3. PaQL: PACKAGE QUERY LANGUAGE

Our PACKAGEBUILDER system extends traditional database functionality to provide full-fledged support for packages. We identify two important reasons to support packages at the database level, rather than at the application level: (a) The data used to construct packages, typically resides in a database system, and packages themselves are structured data objects that naturally should also be stored in and manipulated by a database system. (b) The features of packages and the algorithms for constructing them are not unique to each application; therefore, the burden of package support should be shifted away from the application developers.

We designed PaQL, a SQL-based package query language for PACKAGEBUILDER, which allows for the declarative specification of packages. We use the meal planner scenario from Section 1, to highlight PaQL's key features.

Basic package expressions

```
SELECT    PACKAGE(R) AS P
FROM      Recipes R
```

The introduction of the keyword PACKAGE differentiates PaQL queries from relational SQL queries. Semantically, PACKAGE, constructs *bags* (i.e. *multisets*) from the tuples of the base relations listed within parentheses. With no further constraints, there are infinitely many packages that can be built from non-empty base relations because a package can have duplicate items within it. For example, a recipe can be included in a package many times.

To limit the allowed repetition of tuples from a given relation in a package, users can use the REPEAT keyword, as follows.

```
SELECT    PACKAGE(R) AS P
```

```
FROM        Recipes R REPEAT 0
```

This constructs *sets* from the tuples of the base relation rather than bags. This restricts the package space from infinitely many packages to at most 2^n packages, where n is the number of tuples in the base relation(s). Users can allow up to k duplicates per tuple by specifying REPEAT k, which constricts the package space to $(k+2)^n$.

In the example query, the result is also a relation (not considering repetitions), in that all tuples in the package are drawn from one relation and, hence, have the same schema. However, a package can also be constructed from tuples of different schemata, coming from different relations. To express such a query, the language allows user to list multiple relation within parenthesis, e.g. PACKAGE(R_1, R_2, …, R_k). In this case, the resulting data model may not be relational any more.

Expressing base and global constraints

```
SELECT      PACKAGE(R) AS P
FROM        Recipes R REPEAT 0
WHERE       R.gluten = 'free'
SUCH THAT   COUNT(*) = 3 AND
            SUM(calories) BETWEEN 2000 and 3000
```

A package query defines two kinds of constraints. *Base constraints* are defined in the WHERE clause, as they are semantically equivalent to selection predicates on the base relation(s): any tuple in the package needs to satisfy all base constraints. In the example query, the base constraint R.gluten = 'free' specifies that each meal in the package should be gluten-free.

Global constraints may not be expressed in the WHERE clause, as they indicate properties on sets of tuples rather than on single tuples. They are defined in the SUCH THAT clause: each global constraint needs to be satisfied collectively in the whole package. For example, count(*)=3 specifies that the entire package should have exactly 3 meals.

While base-constraint predicates are in the form of single-tuple predicates (e.g. $Rel.attr \leq const$), global constraints are essentially query-based predicates. For instance, COUNT(*)= 3 is equivalent to (and can be replaced with) (SELECT COUNT(*) FROM P) = 3. Similarly, SUM(calories) BETWEEN 2000 and 3000 can be replaced with (SELECT SUM(calories) FROM P) BETWEEN 2000 and 3000. In general, expressing more complex constraints requires using full-length subqueries, as in the following example:

```
SELECT      PACKAGE(R) AS P
FROM        Recipes R REPEAT 0
WHERE       R.gluten = 'free'
SUCH THAT   COUNT(*) = 3 AND
            SUM(calories) BETWEEN 2000 and 3000 AND
            (SELECT COUNT(*) FROM P
            WHERE carbs > 0) ≥ COUNT(*)/2
```

This query expresses the fact that the athlete wants at least half of the meals to contain carbohydrates.

Expressing optimization objectives

```
SELECT      PACKAGE(R) AS P
FROM        Recipes R REPEAT 0
WHERE       R.gluten = 'free'
SUCH THAT   COUNT(*) = 3 AND
            SUM(calories) BETWEEN 2000 and 3000 AND
            (SELECT COUNT(*) FROM P
            WHERE carbs > 0) ≥ COUNT(*)/2
MAXIMIZE    SUM(protein)
MINIMIZE    SUM(fat)
```

The *objective* clauses MAXIMIZE and MINIMIZE are unique to packages as well: they specify that out of all packages that satisfy the base and global constraints, the ones with larger values in the MAXIMIZE clause and smaller values in the MINIMIZE clause are preferable. An objective clause can list more than one objective, e.g., MAXIMIZE SUM(protein), SUM(carbs).

4. INTERFACE ABSTRACTIONS

Packages queries are more complex, semantically and algorithmically, compared to traditional database queries, and they pose challenges on several fronts: (a) they can have complex specifications, (b) they are hard to process by users given the large volume of results, and (c) they are computationally intensive to evaluate. In this section, we describe several interface abstractions that we implemented in PACKAGEBUILDER to address the first two challenges; we organize the discussion based on three different abstractions. We discuss the evaluation challenge in Section 5.

4.1 Specification

Our *package template* abstraction encodes package specifications in a familiar tabular format (we give a screenshot example in Figure 1). The central component of the template is a sample package, presented as a scrollable table of tuples. Additional components include representations of base and global constraints, optimization objectives, and suggestions for additional package refinements. As a user interacts with the template by highlighting elements in the sample package, PACKAGEBUILDER suggests constraints [7, 1]. For example, when a user selects a cell within the "fats" column, the system infers and proposes several constraints that restrict the amount of fats in each meal, and objectives that minimize the total amount of fats. The package template is quite expressive but is not as powerful as the PaQL language itself. The abstraction tries to strike a balance between ease-of-use and expressive power.

4.2 Presentation

In addition to capturing package specifications, PACKAGEBUILDER presents result packages to users in a way that allows them to meaningfully view the entire package space. PACKAGEBUILDER analyzes the current query specification and selects two dimensions to visually layout the valid packages along. The user can use the visualization to navigate through the available packages by selecting glyphs that represent packages.

4.3 Adaptive exploration

Many users may prefer a trial-and-error, incremental form of package query specification rather than providing a complete and precise specification at the get go. To facilitate this approach, PACKAGEBUILDER presents a sample package that satisfies a few initial constraints. Users can then select good tuples within the sample and request a new sample that replaces the free tuples. Users can repeat this process until they reach the ideal package. PACKAGEBUILDER uses these selections to narrow the search space as well as to identify additional package constraints.

5. QUERY EVALUATION

Evaluating package queries is nontrivial: even if the package does not allow duplicate tuples, the number of possible packages is

in the worst case exponential in the number of base tuples. A brute-force approach that generates and evaluates all candidate packages is thus impractical. We implement three different evaluation strategies: (i) we identify constraints that allow PACKAGEBUILDER to substantially reduce the search space, (ii) we use local search heuristics to discover valid packages, and (iii) we translate package queries into integer programs and use off-the-shelf solvers to find desirable solutions.

5.1 Pruning using cardinality constraints

For simplicity, let us consider package queries involving one base relation. Suppose that n tuples satisfy the base constraints. Our pruning strategy makes upper and lower horizontal cuts on the search lattice. An upper cut at level u prunes away all packages above level u from the search space, corresponding to all packages of cardinality greater than u. Similarly, a lower cut at level l prunes away all packages below level l, corresponding to all packages of cardinality less than l. For queries allowing no duplicates (i.e. with REPEAT 0), these cuts reduce the search space from 2^n to $\binom{n}{l} + \binom{n}{l+1} + \cdots + \binom{n}{u-1} + \binom{n}{u}$. Analogously, the same strategy can reduce the search space from $(k+2)^n$ to $\left(\!\!\binom{n}{l}\!\!\right) + \left(\!\!\binom{n}{l+1}\!\!\right) + \cdots + \left(\!\!\binom{n}{u-1}\!\!\right) + \left(\!\!\binom{n}{u}\!\!\right)$[1] for queries allowing at most k duplicates (i.e. having REPEAT k).

A cardinality constraint is any constraint of the form $a \leq \mathsf{COUNT}(*) \leq b$ which limits the number of tuples in a package to the range $[a,b]$. This trivially prunes the search space with the cuts $u = a$ and $l = b$. Other types of constraints must be transformed into cardinality constraints by exploiting properties of the corresponding aggregation functions. We describe, by the means of simple examples, how SUM can be transformed into cardinality constraints to similarly prune the search space.

Pruning SUM*-based constraints*

Consider the constraint on total calories per package:

$$2000 \leq \mathsf{SUM}(\mathsf{calories}) \leq 3000 \tag{1}$$

Clearly, for any tuple t_i in the base relation, we have that $\mathsf{MIN}(\mathsf{calories}) \leq t_i.\mathsf{calories} \leq \mathsf{MAX}(\mathsf{calories})$. Similarly, for any pair of tuples t_i, t_j, we have that $2{\cdot}\mathsf{MIN}(\mathsf{calories}) \leq t_i.\mathsf{calories} + t_j.\mathsf{calories} \leq 2{\cdot}\mathsf{MAX}(\mathsf{calories})$. Generalizing, for any set of k tuples, $k{\cdot}\mathsf{MIN}(\mathsf{calories}) \leq \mathsf{SUM}(\mathsf{calories}) \leq k{\cdot}\mathsf{MAX}(\mathsf{calories})$. This entails that

$$\begin{aligned} l &= \lceil \tfrac{2000}{\mathsf{MAX}(\mathsf{calories})} \rceil \\ u &= \lfloor \tfrac{3000}{\mathsf{MIN}(\mathsf{calories})} \rfloor \end{aligned}$$

are the least and greatest number of tuples that can possibly satisfy constraint (1). In fact, we can achieve the lower bound of the constraint by having l recipes with $\mathsf{MAX}(\mathsf{calories})$ each, and we can achieve the upper bound of the constraint by having at most u recipes with $\mathsf{MIN}(\mathsf{calories})$ each.

This strategy applies to queries generating multisets of any kind, i.e., having no REPEAT keyword, or having REPEAT k, for all k. However, the bounds l and u can be too loose in cases where REPEAT k is defined, especially for low k's. Assume, for instance, that the values of calories are as follows:

tuple	calories
t_1	600
t_2	750
t_3	800
t_4	1000
t_5	4000

Here, $\mathsf{MIN}(\mathsf{calories}) = 600$ and $\mathsf{MAX}(\mathsf{calories}) = 4000$. Hence $l = 1$ and $u = 5$, which only prunes away the empty package from the search space. However, if no duplicates are allowed, no package of size 1, 2, 4 or 5 can actually achieve a sum in the range of constraint (1). Similarly, if at most two duplicates are allowed, there are solutions of size 2 (i.e. $\{t_4, t_4\}$). With three duplicates allowed, there are also solutions of size 4, and so on.

With k duplicates allowed, the tightest lower and upper cardinality bounds are computed as follows. Let $\mathsf{MIN}_k(\mathsf{calories})$ and $\mathsf{MAX}_k(\mathsf{calories})$ be the prefix sum and the inverse prefix sum[2] of the sorted sequence of calories values where each element is duplicated exactly k times. For example, for the sequence $2, 5, 8$ and $k = 3$, MIN_k and MAX_k are the prefix sum and the inverse prefix sum of the sequence $2, 2, 2, 5, 5, 5, 8, 8, 8$, respectively. Furthermore, consider two auxiliary functions L and U, such that $L(2000, \mathsf{MAX}_k)$ calculates the position of the first element in the sequence MAX_k whose calories is at least 2000, and $U(3000, \mathsf{MIN}_k)$ calculates the position of the last element in the sequence MIN_k whose calories is not above 3000. The lower and upper cardinality cuts are:

$$\begin{aligned} l &= L(2000, \mathsf{MAX}_k(\mathsf{calories})) \\ u &= U(3000, \mathsf{MIN}_k(\mathsf{calories})) \end{aligned}$$

It can be shown that the bounds computed with the prefix sums are equal to or tighter than those computed only with MIN and MAX, or, in other words, that $L(2000, \mathsf{MAX}_k(\mathsf{calories})) \geq \lceil \frac{2000}{\mathsf{MAX}(\mathsf{calories})} \rceil$ and $U(3000, \mathsf{MIN}_k(\mathsf{calories})) \leq \lfloor \frac{3000}{\mathsf{MIN}(\mathsf{calories})} \rfloor$. In the previous example, the new bounds would be $l = 1$, $u = 3$ if no repetition are allowed, which is better than those provided by MIN and MAX alone because they prune out a larger portion of the search space. However, the two methods present complexity tradeoffs, in that computing prefix sums on the sorted sequence is more expensive, in general, than computing only the minimum and maximum elements.

5.2 Heuristic Local Search

Pruning often reduces the search space significantly, but this reduction alone is seldom sufficient. In addition to pruning algorithms, PACKAGEBUILDER employs heuristics that start from a given candidate package P_0, and perform local search to generate new packages iteratively. Given the package P_0, PACKAGEBUILDER identifies tuple exchanges that can lead to a valid package by constructing a single SQL query to generate all possible replacements. For example, suppose we wish to generate meal packages with less than 3,000 total calories. Given the starting package P_0 that has a total of 3,500 calories, we can look for all possible single-tuple replacements with the following SQL query:

```
SELECT    T1.id,R1.id
FROM      P0 AS T1, Recipes AS R1
WHERE     3500 - T1.calories + R1.calories ≤ 3000
```

The query is a selection over a cartesian product between the candidate package and the recipe relation. This approach is very efficient if we are looking to replace only a few tuples at a time.

[1] We remind the readers that $\left(\!\!\binom{n}{k}\!\!\right) = \binom{n+k-1}{k}$ denotes the number of combination with repetitions, i.e. the number of multisubsets of size k that can be formed from a set of size n.

[2] We consider the inverse prefix sum as the prefix sum of the reversed sequence. For instance, the inverse prefix sum of $2, 6, 10$ is $10, 16, 18$.

For k replacements, however, this method would require a $2k$-way join, which quickly becomes intractable. This method of heuristic search is particularly useful for adaptive exploration (Section 4.3), where users usually request the replacement of only a few tuples.

5.3 Constraint Solvers

For certain classes of package queries, PACKAGEBUILDER employs state-of-the-art constraint solvers to derive valid packages. PACKAGEBUILDER translates the package query to integer programs, with an approach similar to the Tiresias system [8]. After expressing the package query as an integer program, PACKAGEBUILDER invokes off-the-shelf constraint solvers to derive a solution.

5.4 Considerations

Our experience with PACKAGEBUILDER shows that each of the evaluation techniques (pruning, heuristics, and IP reduction) have different strengths and weaknesses. Heuristic search is very effective if the constraints of the package query are not very restrictive, and the search can navigate through different solutions with small steps (single-tuple swaps). However, in a heavily constrained space, heuristic search may have trouble reaching a valid package from an invalid package. Constraint solvers can handle fairly complex queries, with a large number of variables and constraints. However, perhaps surprisingly, they do not seem to perform as well in smaller problems. They are also limited to returning a single package solution at a time, and getting more packages requires subsequent reruns. Given the large variation and different characteristics of the three approaches, there is no clear winner, and in its current version, PACKAGEBUILDER combines all three evaluation methods to efficiently derive packages.

6. REFERENCES

[1] A. Abouzied et al. Dataplay: Interactive tweaking and example-driven correction of graphical database queries. In *UIST*. ACM, 2012.

[2] A. Parameswaran et al. Recommendation systems with complex constraints: A courserank perspective. *TOIS*, June 2011.

[3] C. Olsten et al. Viqing: Visual interactive querying. In *Symposium on Visual Languages*, pages 162–169. IEEE, 1998.

[4] C. Stolte et al. Polaris: a system for query, analysis, and visualization of multidimensional databases. *CACM*, 51, Nov. 2008.

[5] D. Chen et al. *Applied integer programming: modeling and solution*. Wiley. com, 2011.

[6] S. J. Russell et al. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995.

[7] S. Kandel et al. Wrangler: interactive visual specification of data transformation scripts. In *CHI*. ACM, 2011.

[8] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *SIGMOD*, pages 337–348. ACM, 2012.

[9] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.