Chapter 1.4.2
Bulk Types: Built-In or Add-On?

Florian Matthes and Joachim W. Schmidt

Technical University Hamburg-Harburg
Harburger Schloßstraße 20
D-21071 Hamburg, Germany

Summary This text is a summary of [8].

Bulk structures play a central rôle in data-intensive application programming. The issues of *bulk type* definition and implementation as well as their integration into database programming languages are, therefore, key topics in current DBPL research.

In this paper we raise a more general language design and implementation issue by asking whether there should be at all built-in bulk types in DBPLs. Instead, one could argue tat bulk types should be realized exclusively as user-definable add-s to unbiased core languages with appropriate primitives and abstraction facilities.

In searching for an answer we first distinguish two substantially different levels on which bulk types are supported. *Elementary Bulk* essentially copes with persistent storage of mass data, their identification and update. *Advanced Bulk* provides additional support required for data-intensive applications such as optimized associative queries and integrity support under concurrency and failure.

Our long-term experience with bulk types in the DBPL language and system clearly shows the limitation of the built-in approach: built-in *Advanced Bulk*, as elaborate as it may be, frequently does not cover the whole range of demands of a fully-fledged application and often does not provide a decent pay-off for its implementation effort. On the other hand, restriction to built-in *Elementary Bulk* gives too little user-support for most data-intensive applications.

Irrespective of the particular kind of bulk data structures present in a given database programming environment (e.g. lists, sets, relations in traditional DBPLs; class extents in object-oriented databases; base predicates in deductive databases or extensionally defined functions in functional databases), one can distinguish two fundamentally different approaches to bulk type support within a language framework:

Built-In Bulk Types are provided as first-class parameterized type constructors in several programming languages [13, 6, 12, 11], their syntax, type rules, semantics and implementation being hard-wired into the language processor and the run-time system support.

Add-On Bulk Types are defined and implemented utilizing standard built-in language mechanisms (typing, naming, binding, scoping or recursion) of a sufficiently generic general-purpose base language like ML [10], Modula-3 [3], Napier-88 [4] or Eiffel [9].

In this paper we analyze the motivations and basic assumptions behind both (extreme) approaches and explore their individual advantages and shortcomings. This investigation is based on the one hand on our substantial experience with the design and implementation of database programming languages incorporating powerful built-in bulk type support [13, 5, 7]. On the other hand it reports on the design rationale for Tycoon[1], a language and system environment with add- bulk types.

[1] Tycoon: Typed Communicatng Objects in Open Environments.

The issue of built-in vs. add- bulk types has a longer tradition in the Persistent Programming Language community, in the DBPL camp, however, the main goal always consisted in building in advanced bulk types. By reporting on our own past experiences and future expectations we attempt to contribute to a more open and objective discussion on this important DBPL research issue.

We report on experience with built-in bulk types by referring essentially to DBPL, a set- and predicate-based procedural database programming language [14]. We illustrate issues in language design, system construction and application programming involving built-in bulk types.

The numerous requirements for bulk types (see, e.g., [1, 2]) can be classified into *elementary* and *advanced* requirements which we will sumarize below. This discussion is valid for both, built-in as well as add- bulk types, and, therefore, also establishes a framework for an investigation of environments that enable their users to add-on bulk types meeting these elementary and advanced requirements.

As a first cut, the distinction between an elementary and an advanced requirement can be based on the kind of technology that is required to provide a particular bulk type support. Elementary requirements usually can be met by *local* modifications to language processors (compilers, abstract machines, run-time support libraries) essentially based on *programming language* technology. On the other hand, more advanced requirements, for example effective query optimization or stratification analysis of fixed-point queries, involve *global* analysis and modifications as provided by advanced *database* technology.

The built-in approach can be characterized by the following assumptions:

– All language implementation details are hidden from the application programmer (How is a bulk type implemented? How is a query represented, transformed and evaluated? How is serializability achieved?)
– The efficient implementation of the database language depends crucially on detailed language design time knowledge about permissible language constructs: Which (bulk) types exist? What is the syntax of the query (sub-)language? What are the algebraic properties of the built-in language primitives like equality predicates, comparison operations, aggregate functions?
– Significant optimizations are based on a careful case-analysis within the available language space. Examples are optimizations for special cases like flat relations, unordered collections, read-only transactions or one variable selection queries.

Some of these assumptions are in conflict with the programming language principles of orthogonality and free extensibility. Accordingly, much work in the database community aims at *extending database technology* to support, e.g.

– new aggregate functions in queries,
– new operations within queries (e.g. test for intersection of geometric objects),
– new bulk structures (e.g. ordered sequences),
– new operations in the target list of query expressions,
– new storage structures,
– new join algorithms, etc.

Despite progress made with extensible database systems [15], none of today's database programming language implementations provides adequate support for the above extensions.

The paper outlines the initial design of the Tycoon bulk type environment that aims at defining uniformly and implementing systematically declarative iteration abstractions applicable to a wide variety of bulk structures. Specifically, we explain how the concepts of data independence and query optimization can be generalized to liberally extensible systems. We also identify specific language mechanisms required for the efficient and robust construction of such open and extensible database application systems.

In our experience the main arguments against the traditional built-in approach to bulk types can be summarized as follows:

Reusability: It is difficult (often impossible) to safely re-use existing system components of the built-in system environment (buffer manager, polymorphic index structures, wait-for-graph management, clustering algorithms etc.).

Scalability: It is typically not possible to eliminate unnecessary functionality from the built-in system environment (concurrency control or recovery actions, components for recursive query evaluation).

Adaptability: Even if one has access to the implementations of the built-in bulk structures, modifications of these components (e.g. replacing B-Tree index structures by hashed structures or replacing a garbage collection algorithm) have to be performed in a "lower level" programming language with obvious negative consequences on the overall system correctness and long-term system evolution.

The apparent resistance of current DBPL *system implementations* against DBPL *language extensions* gives rise to the following important research question. Can one isolate "generic" language constructs that aid in the systematic construction of flexible DBPL processors and meet more than just a limited set of structures and algorithms foreseen at language design time? The second part of this paper addresses the relationship between DBPL language and system extensibility.

Recent advances towards expressive type systems and highly effective compilation schemes may allow the *elementary* requirements for bulk data storage (including persistence management) to be satisfied without resorting to the built-in approach. Given state-of-the-art language technology, a high investment in built-in bulk types (as exemplified by DBPL) can only be justified by the support it provides for *advanced* requirements like generalized associative queries or integrity constraint management in multi-user environments. We finally reported on our current work that aims to satisfy even these advanced requirements by user-definable add-ons to generic core languages focussing on linguistic and architectural aspects, since important component technology seems to be already available in extensible database systems.

Clearly there remain many open research issues for further theoretical and experimental work. What is a suitable formal framework for data models supporting generalized bulk types? Is it necessary to equip DBPLs with specific *syntactic* support for a concise notation of bulk data manipulations (e.g. list comprehensions)? Is there a taxonomy for general optimization strategies (algebraic vs. operational, static vs. dynamic, deterministic vs. probabilistic)? Where should the meta-information required for optimizations come from (static program analysis, program annotations via pragmas, access to program specifications, compile-time or run-time reflection)? Can standardized iteration abstractions be exploited for program verification tasks?

Acknowledgement This work was supported by the European Commission under ESPRIT BRA contract # 3070 (FIDE).

References

1. M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Deductive and Object-oriented Databases.* Elsevier Science Publishers, Amsterdam, Netherlands, 1990.
2. M. Atkinson, P. Richard, and P. Trinder. Bulk types for large scale programming. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.
3. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report ORC-1, Olivetti Research Center, 2882 Sand Hill Road, Menlo Park, California, 1988.
4. A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88 – a database programming language? In *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, June 1989.
5. J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, and C.A. Zehnder. Modula/R report, lilith version. Technical report, Department Informatik, ETH Zürich, Switzerland, February 1983.
6. C. Lécluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, Illinois*, pages 424–433, 1988.
7. F. Matthes and J.W. Schmidt. The type system of DBPL. In *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, pages 255–260, June 1989.
8. F. Matthes and J.W. Schmidt. Bulk types: Built-in or add-on? In *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.
9. B. Meyer. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1988.
10. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.
11. S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.
12. A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 46–57, 1989.
13. J.W. Schmidt. Some high level language constructs for data of type relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.
14. J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1988.
15. M. Stonebraker. Special issue on database prototype systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.