

Chapter 9: Languages

Introduced by Joe Hellerstein

Selected Readings:

Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), 1977, 247-261.

Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2), 2006, 121-142.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. *CIDR*, 2011.

From reading database papers, you might expect that typical database users are data analysts, business decision-makers, or IT staff. In practice, the majority of database users are software engineers, who build database-backed applications that are used further up the stack. Although SQL was originally designed with non-technical users in mind, it is rare for people to interact directly with a database via a language like SQL unless they are coding up a database-backed application.

So if database systems are mostly just APIs for software development, what do they offer programmers? Like most good software, database systems offer powerful abstractions.

Two stand out:

1. The transaction model gives programmers the abstraction of a single-process, sequential machine that never fails mid-task. This protects programmers from a gigantic cliff of complexity—namely, the innate parallelism of modern computing. A single compute rack today has thousands of cores running in parallel across dozens of machines that can fail independently. Yet application programmers can still blithely write sequential code as if it were 1965 and they were loading a deck of punchcards into a mainframe to be executed individually from start to finish.
2. Declarative query languages like SQL provide programmers with abstractions for manipulating sets of data. Famously, declarative languages shield programmers from thinking about *how* to access data items, and instead let them focus on *what* data items to return. This data independence also shields application programmers from changes in the organization of underlying databases, and shields database administrators from getting involved in the design and maintenance of applications.

Just how useful have these abstractions been over time?

How useful are they today?

1. As a programming construct, serializable transactions have been very influential. It is easy for programmers to bracket their code with BEGIN and COMMIT/ROLLBACK. Unfortunately, as we discussed in Chapter 6, transactions are expensive, and are often compromised. “Relaxed” transactional semantics break the serial abstraction for users and expose application logic to the potential for races and/or unpredictable exceptions. If application developers want to account for this, they have to manage the complexity of concurrency, failure, and distributed state. A common response to the lack of transactions is to aspire to “eventual consistency” [21], as we discuss in the weak isolation section. But as we discussed in Chapter 6, this still shifts all the correctness burdens to the application developer. In my opinion, this situation represents a major crisis in modern software development.
2. Declarative query languages have also been a success—certainly an improvement over the navigational languages that preceded them, which led to spaghetti code that needed to be rewritten every time you reorganized the database. Unfortunately, query languages are quite different from the imperative languages that programmers usually use. Query languages consume and produce simple unordered “collection types” (sets, relations, streams); programming languages typically carry out ordered execution of instructions, often over complex structured data types (trees, hashtables, etc.). Programmers of database applications are forced to bridge this so-called “impedance mismatch” between programs and database queries. This has been a hassle for database programmers since the earliest days of relational databases.

Database Language Embeddings: Pascal/R

The first paper in this section presents a classical example of tackling the second problem: helping imperative programmers with the impedance mismatch. The paper begins by defining operations for what we might now recognize (40-odd years later!) as familiar collection types: the “dictionary” type in Python, the “map” type in Java or Ruby, etc. The paper then patiently takes us through the possibilities and pitfalls of various language constructs that seem to recur in applications across the decades. A key theme is a desire for differentiating between enumeration (for generating output) and quantification (for checking properties)—the latter can often be optimized if you are explicit. In the end, the paper proposes a declarative, SQL-like sublanguage for Relation types that is embedded into Pascal. The result is relatively natural and not unlike some of the better interfaces today.

Although this approach seems natural now, the topic took decades to gain popular attention. Along the way, database “connectivity” APIs like ODBC and JDBC arose as a crutch for C/C++ and Java—they allowed users to push queries to the DBMS and iterate through results, but the type systems remained separate, and bridging from SQL types to host language types was unpleasant. Perhaps the best modern evolution of ideas like Pascal/R is Microsoft’s LINQ library, which provides language-embedded collection types and functions so application developers can write query-like code over a variety of backend databases and other collections (XML documents, spreadsheets, etc.) We included a taste of LINQ syntax in the DryadLINQ paper in Chapter 5.

In the 2000’s, web applications like social media, online forums, interactive chat, photo sharing and product catalogs were implemented and reimplemented over relational database backends. Modern scripting languages for web programming were a bit more convenient than Pascal, and typically included decent collection types. In that environment, application developers eventually saw recognized patterns in their code and codified them into what are now called Object-Relational Mappings (ORMs). Ruby on Rails was one of the most influential ORM to begin with, though by now there are many others. Every popular application programming language has at least one, and there are variations in features and philosophy. The interested reader is referred to Wikipedia’s “List of object-relational mapping software” wiki.

ORMs do a few handy things for the web programmer. First they provide language-native primitives for working with collections much like Pascal/R. Second they can enable updates to in-memory language objects to be transparently reflected in the database-backed state. They often offer some language-native syntax for familiar database design concepts

like entities, relationships, keys and foreign keys. Finally, some ORM including Rails offer nice tools for tracking the way that database schemas evolve over time to reflect changes in the application code (“migrations” in Rails terminology).

This is an area where the database research community and industry should pay more attention: these are our users! There are some surprising—and occasionally disconcerting—disconnects between ORM and databases [6]. The author of Rails, for example, is a colorful character named David Heinemeier Hansson (“DHH”) who believes in “opinionated software” (that reflects his opinions, of course). He was quoted saying the following:

I don’t want my database to be clever! ...I consider stored procedures and constraints vile and reckless destroyers of coherence. No, Mr. Database, you can not have my business logic. Your procedural ambitions will bear no fruit and you’ll have to pry that logic from my dead, cold object-oriented hands . . . I want a single layer of cleverness: My domain model.

This unwillingness to trust the DBMS leads to many problems in Rails applications. Applications written against ORM are often very slow—the ORM themselves don’t do much to optimize the way that queries are generated. Instead, Rails programmers often need to learn to program “differently” to encourage Rails to generate efficient SQL—similar to the discussion in the Pascal/R paper, they need to learn to avoid looping and table-at-a-time iteration. A typical evolution in a Rails app is to write it naively, observe slow performance, study the SQL logs it generates, and rewrite the app to convince the ORM to generate “better” SQL. Recent work by Cheung and colleagues explores the idea that program synthesis techniques can automatically generate these optimizations [9]; it’s an interesting direction, and time will tell how much complexity it can automate away. The separation between database and applications can also have negative effects for correctness. For example, Bailis recently showed [6] how a host of existing open-source Rails applications are susceptible to integrity violations due to improper enforcement within the application (instead of the database).

Despite some blind spots, ORM have generally been an important practical leap forward in programmability of database-backed applications, and a validation of ideas that go back as far as Pascal/R. Some good ideas take time to catch on.

Stream Queries: CQL

Our second paper on CQL is a different flavor of language work—it’s a query language design paper. It presents

the design of a new declarative query language for a data model of streams. The paper is interesting for a few reasons. First, it is a clean, readable, and relatively modern example of query language design. Every few years a group of people emerges with yet another data model and query language: examples include Objects and OQL, XML and XQuery, or RDF and SPARQL. Most of these exercises begin with an assertion that “X changes everything” for some data model X, leading to the presentation of a new query language that often seems familiar and yet strangely different than SQL. CQL is a refreshing example of language design because it does the opposite: it highlights that fact that streaming data, viewed through the right lens, actually changes very little. CQL evolves SQL just enough to isolate the key distinctions between querying “resting” tables and “moving” streams. This leaves us with a crisp understanding of what’s really different, semantically, when you have to talk about streams; many other current streaming languages are quite a bit more ad hoc and messy than CQL.

In addition to this paper being a nice exemplar of thoughtful query language design, it also represents a research area that received a lot of attention in the database literature, and remains intriguing in practice. The first generation of streaming data research systems from the early 2000’s [1, 17, 16, 8] did not have major uptake either as open source or in the variety of startups that grew out of those systems. However the topic of stream queries has been gaining interest again in industry in recent years, with open source systems like SparkStreaming, Storm and Heron seeing uptake, and companies like Google espousing the importance of continuous dataflow as a new reality of modern services [2]. We may yet see stream query systems occupy more than their current small niche in financial services.

Another reason CQL is interesting is that streams are something of a middle ground between databases and “events”. Databases store and retrieve collection types; event systems transmit and handle discrete events. Once you view your events as data, then event programming and stream programming look quite similar. Given that event programming is a widely-used programming model in certain domains (e.g. Javascript for user interfaces, Erlang for distributed systems), there should be a relatively small impedance mismatch between an event programming language like Javascript and a data stream system. An interesting example of this approach is the Rx (Reactive extensions) language, which is a streaming addition to LINQ that makes programming event streams feel like writing functional query plans; or as its author Erik Meijer puts it, “your mouse is a database” [15].

Programming Correct Applications without Transactions: Bloom

The third paper on Bloom connects to a number of the points above; it has a relational model of state at the application level, and a notion of network channels that relates to CQL streams. But the primary goal is to help programmers manage the loss of the first abstraction at the beginning of this chapter introduction; the one I described as a major crisis. A big question for modern developers is this: can you find a correct distributed implementation for your program without using transactions or other other expensive schemes to control orders of operation?

Bloom’s answer to this question is to give programmers a “disorderly” programming language: one that discourages them from accidentally using ordering. Bloom’s default data structures are relations; its basic programming constructs are logical rules that can run in any order. In short, it’s a general-purpose language that is similar to a relational query language. For the same reason that SQL queries can be optimized and parallelized without changing output, simple Bloom programs have a well-defined (consistent) result independent of the order of execution. The exception to this intuition comes with lines of Bloom code that are “non-monotonic”, testing for a property that can oscillate between true and false as time passes (e.g. “NOT EXISTS x” or “HAVING COUNT() = x”.) These rules are sensitive to execution and message ordering, and need to be “protected” by coordination mechanisms.

The CALM theorem formalizes this notion, answering the question above definitively: you can find a consistent, distributed, coordination-free implementation for your program if and only if its specification is monotonic [13, 5]. The Bloom paper also illustrates how a compiler can use CALM in practice to pinpoint the need for coordination in Bloom programs. CALM analysis can also be applied to data stream languages in systems like Storm with the help of programmer annotations [3]. A survey of the theoretical results in this area is given in [4].

There has been a flurry of related language work on avoiding coordination: a number of papers propose using associative, commutative, idempotent operations [12, 19, 10]; these are inherently monotonic. Another set of work examines alternative correctness criteria, e.g., ensuring only specific invariants over database state [7] or using alternative program analysis to deliver serializable outcomes without implementing traditional read-write concurrency [18]. The area is still new; papers have different models (e.g. some with transactional boundaries and some not) and often don’t agree on definitions of “consistency” or “coordination”. (CALM defines consistency in terms of globally de-

terministic outcomes, coordination as messaging that is required regardless of data partitioning or replication [5].) It's important to get more clarity and ideas here—if programmers can't have transactions then they need help at the app-development layer.

Bloom also serves as an example of a recurring theme in database research: general-purpose declarative languages (a.k.a. “logic programming”). Datalog is the standard example, and has a long and controversial history in database research. A favorite topic of database theoreticians in the 1980's, Datalog met ferocious backlash from systems re-

searchers of the day as being irrelevant in practice [20]. More recently it has gotten some attention from (younger) researchers in databases and other applied fields [11]—for example, Nicira's software-defined networking stack (acquired by VMWare for a cool billion dollars) uses a Datalog language for network forwarding state [14]. There is a spectrum between using declarative sublanguages for accessing database state, and very aggressive uses of declarative programming like Bloom for specifying application logic. Time will tell how this declarative-imperative boundary shifts for programmers in various contexts, including infrastructure, applications, web clients and mobile devices.

References

- [1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal/The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.
- [2] T. Akidau et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *VLDB*, 2015.
- [3] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 52–63. IEEE, 2014.
- [4] T. J. Ameloot. Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. *ACM SIGMOD Record*, 43(2):5–16, 2014.
- [5] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *Journal of the ACM (JACM)*, 60(2):15, 2013.
- [6] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral Concurrency Control: An empirical investigation of modern application integrity. In *SIGMOD*, 2015.
- [7] P. Bailis, A. Fekete, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Coordination avoidance in database systems. In *VLDB*, 2015.
- [8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, et al. Telegraphicq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
- [9] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.
- [10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):10, 2015.
- [11] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.
- [12] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.
- [13] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *ACM SIGMOD Record*, 39(1):5–19, 2010.
- [14] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, 2014.

- [15] E. Meijer. Your mouse is a database. *Queue*, 10(3):20, 2012.
- [16] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.
- [17] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, et al. The niagara internet query system. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.
- [18] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.
- [19] M. Shapiro, N. Prego, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. INRIA TR 7506, 2011.
- [20] M. Stonebraker and E. Neuhold. The laguna beach report. Technical Report 1, International Institute of Computer Science, 1989.
- [21] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, et al. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.