Beyond Isolation: Research Opportunities in Declarative Data-Driven Coordination

Lucja Kot, Nitin Gupta, Sudip Roy, Johannes Gehrke, and Christoph Koch

Cornell University
Ithaca, NY 14853, USA
{lucja, niting, sudip, johannes, koch}@cs.cornell.edu

ABSTRACT

There are many database applications that require users to coordinate and communicate. Friends want to coordinate travel plans, students want to jointly enroll in the same set of courses, and busy professionals want to coordinate their schedules. These tasks are difficult to program using existing abstractions provided by database systems because in addition to the traditional ACID properties provided by the system they all require some type of *coordination* between users. This is fundamentally incompatible with isolation in the classical ACID properties.

In this position paper, we argue that it is time for the database community to look beyond isolation towards principled and elegant abstractions that allow for communication and coordination between some notion of (suitably generalized) transactions. This new area of *declarative data-driven coordination* (D3C) is motivated by many novel applications and is full of challenging research problems. We survey existing abstractions in database systems and explain why they are insufficient for D3C, and we outline a plethora of exciting research problems.

1. INTRODUCTION

Every sin is the result of collaboration. — Stephen Crane

1.1 Databases and Social Applications

In his book "The Seven Habits of Highly Effective People," Stephen Covey outlines seven inspirational habits that lead the reader on a path to maturing from dependence to independence and finally to interdependence. As people become interdependent, they start working together to achieve a common goal through coordination and collaboration. Do databases need to go a similar path and move beyond independence?

In the last decade, the amount of data on the Internet has grown at a staggering rate. The rise of Web 2.0 has fueled that growth to no small extent by providing platforms that enable people to create, upload, and share content very easily. However, people do much more than just produce and consume data; they are beginning to center their life around the Web and use social applications for very complex data-driven tasks.

As a concrete, seemingly mundane example of a social application, consider a simple travel booking application. Our community is very familiar with these as they have involved databases for many years. Yet all of us have faced the scenario where we wished to not just book travel tickets for ourselves, but to *coordinate* travel plans with family, friends or colleagues. How does this coordination happen today? Typically, it starts with a lengthy phone or email conversation to decide on the itinerary. Then, one person books tickets for everyone, and there is another round of emails to sort out the finances. Or everybody arranges to book at about the same time, hoping that the airline seats do not fill up in the meanwhile.

Coordinating travel plans manually is not just inefficient, it is unsatisfying from a system design standpoint. Certainly it is possible to build the application that handles this travel booking scenario; we just need to find some guys in a garage to code it up perhaps using triggers, nested transactions or special-purpose application code and data structures. (We will discuss these potential implementations shortly.) However, it is worrisome that such a simple use case like cooperative travel booking, which has motivated so much database research (and database exam questions), today still requires ad-hoc "hacks" for this conceptually simple and useful functionality.

The fact that modern databases are difficult to use in collaborative settings is not an accident. The requirements from the social application of coordinating some actions clashes with a fundamental abstraction that is a cornerstone of the database community: the ACID transaction. Transactions were designed to be isolated from each other; therefore, the transaction abstraction provides no mechanism for coordination. In social applications, transactions are still a meaningful concept, and atomicity and durability are still crucial. Isolation, however, may be too strict a requirement and in the use case above has become an obstacle that the application programmer has to navigate around.

Coordinating transactions would certainly facilitate basic

coordination with a small group of friends on travel bookings, leisure activities and course enrollments [5]. However, there are many other applications that would benefit from various types of transaction coordination.

For example, consider a professor who wishes to schedule a weekly meeting with each advisee, ensuring that everyone's availability is respected and that no two meetings fall in the same slot. More generally, people frequently want to coordinate individual contributions to meet a given goal. Some simple examples are wedding gift purchases or potluck dinner planning to ensure a balanced meal. In a more serious vein, grass-roots groups frequently want to organize coordinated events, whether in the aftermath of a disaster, for social or environmental activism, or other reasons. Some social networking sites today already enable a limited form of volunteer coordination, but clearly much more could be achieved with a suitable technological solution.

Sometimes, users want to agree on more than just binary yes/no decisions relating to participation or contributions. Charities run fundraising drives which include gift matching pledges to encourage more donations. If multiple charities are involved in a single drive, the donation coordination problem can become quite complex [2].

Support for complex coordination would also be very useful in massively multiplayer online games. These games typically involve thousands of players, most of them unknown to each other. Groups of players often want to work together to achieve certain goals. A mechanism allowing users to communicate and coordinate plans of action with partners would significantly enhance the gameplay experience, particularly if it allowed partnerships to be formed on-the-fly based on shared player goals.

It is clear that data-driven coordination is applicable to a wide variety of domains and can take many forms. However, to date, coordination has not been recognized as fundamental to a wide range of data-intensive applications, and has only been implemented using ad-hoc solutions.

1.2 Life Beyond Isolation

We believe that it is time to change this pitiful situation. We need research into *declarative data-driven coordination* (D3C), abstractions that extend transactions beyond isolation to allow some form of information flow. We postulate that the fundamental design principle behind D3C should be that in the data-driven setting, coordination should be expressed in a *declarative* way. That is, programmers should only specify *what* coordination is needed, not *how* it should be achieved. Declarativity has long been an underlying design principle in databases, and we believe that it is as applicable to coordination as it is to querying. The complexity of dealing with details of concurrency and coordination should be moved away from the programmer and into the system, where it belongs.

In a system designed to support D3C, our travel booking coordination scenario might play out as follows. Suppose a user, Catherine, wants to travel to Paris on the same flight as her friend Sylvia. Rather than communicating out-of-band via email or phone, Catherine would simply specify her database update as follows, in an SQL-like language or in a suitable visual interface:

```
COMMUNICATE flightno IN
INSERT INTO booking
SELECT 'Catherine', flightno
FROM flights
WHERE destination = 'Paris'
COORDINATING WITH
SELECT 'Sylvia', flightno
FROM flights
WHERE destination = 'Paris'
```

Assuming Sylvia wants to coordinate with Catherine, she would issue a symmetric update with the strings 'Catherine' and 'Sylvia' exchanged. Each statement asks to book a seat on a single but arbitrary flight to Paris on which the other friend also books a flight. However, this constraint across updates is not equivalent to a WHERE-clause, which in this case would never allow either friend to book a flight. Instead, coordination on flightno works like a postcondition on the updates: the updates are only allowed to fire if the execution of both updates will cause the coordination constraint to be satisfied right after the updates.

The system would recognize that the two transactions want to coordinate, and make bookings for each user accordingly. The transactions would remain separate and isolated except for the single information exchange about the flight number. With multiple flights to choose from and no further constraints given, there is a degree of nondeterminism here; the system has to choose one suitable flight and make reservations for both friends.

The principle of coordination by queries and updates just sketched raises many research questions. The goal of the remainder of this article is to voice many of them, and to outline a research program with the aim of achieving the vision of declarative data-driven coordination.

2. D3C AND MODERN DATABASES

Although isolation has always been fundamental to transactions, database research has taken some steps towards enabling coordination over the years. A few abstractions allow a limited form of information exchange among transactions. In addition, some database mechanisms can be used to implement forms of coordination, although they were not originally designed for this purpose.

The best-known abstraction that allows some form of communcation between transactions is nested transactions [7]. In nested transactions, inner transactions can pass information to the outer transaction using variables. However, this model is very restricted in that it does not permit bidirectional information flow between inner transactions. The type of coordination shown in our previous example – Cather-

ine and Sylvia's flight booking – requires mutual information exchange in both directions: Catherine needs to select the same flight number as Sylvia, and both these decisions need to made at the same time as otherwise there may be no longer a seat available after booking the first seat. Thus nested transactions unfortunately cannot be used to implement coordination.

Triggers are an alternate mechanism that might be used to implement coordination [13]. However, triggers were not originally designed for this purpose, and therefore fall short in many ways as a solution for D3C. First, like nested transactions, they do not easily enable bidirectional information flow between transactions. Second, triggers are designed to *initiate* transactions. No related constructs or abstractions exist to "pause" transactions while their progress depends on some other activity in the system, as would be needed in many of our D3C scenarios. Finally, triggers are notoriousy difficult to manage if orderly and controlled behavior is needed. Maintaining transactional properties like atomicity and durability as well as handling the consequences of decisions about committing and aborting would become highly complex in a trigger-based system.

Since coordination is currently not supported at the database level, today's developers must implement it in application code. Such an implementation must include all the coordination mechanisms, which are nontrivial to design and program. These mechanisms need to make heavy use of the underlying database as they involve database state, and they themselves need to persist data in case a coordination needs to wait to take place in the future. Thus implementing coordination at the application level creates a tight coupling between middle and database tier, a suboptimal design. In addition, implementing coordination at the middle tier may remove opportunities for optimization of the coordination within the database.

Coordination Beyond Databases. Our community's thinking which is solely centered on isolation is in contrast with other fields of computer science, which have long permitted, supported, and made good use of communication between concurrent tasks. For example, operating systems typically provide low-level mechanisms such as message passing, shared memory, locks, and semaphores that enable programs to coordinate their execution. There are also efficient higher-level models of coordination between programs, such as transactional memory [6].

Formalisms like the Pi-calculus [8] provide abstractions such as *channels* between processes; these abstractions allow to model and reason about the execution of communicating programs. Channels and other communication methods are also available to programmers as implementation mechanisms. Many programming languages already come with concurrency support – to name only a few, Concurrent ML [9], Erlang [11], Stackless Python [1] and Concurrent Haskell [4]. Cmmunication also plays an important role in multiagent systems [12] where it facilitates tasks such as distributed planning and decision making.

For many years, the database community was able to get away with a comparative lack of attention to communication. Most traditional applications did not require it, and assuming full isolation allowed for design simplifications and optimizations. However, modern collaborative data-driven tasks expose the limitations of full isolation among transactions. As a community, it is time for us to expand our horizons and provide a mechanism for coordination among transactions; we need solutions to support D3C! However, as we shall see, moving from an isolation-only model of transactions to D3C raises many exciting research problems touching all aspects of a database system.

3. RESEARCH CHALLENGES

In this section, we present specific technical challenges associated with supporting D3C. Our presentation is organized to match the flow of a design process for a full end-to-end solution. We start by discussing desiderata for the coordination model itself and for its efficient implementation. Next, we move on to issues that arise when integrating coordination with other database functionality such as atomic transactions and user privacy. Finally, we give some thought to architecting a database system with support for coordination.

3.1 Devising a Coordination Abstraction

Declarative Abstraction. As we have already mentioned in the introduction, the very first research challenge is to design a clean, powerful and expressive declarative model for coordination. This model should abstract away from the implementation of coordination and require users to only specify *what* coordination is required instead of specifying *how* coordination is achieved. Designing such a model will not be easy; we have already seen that there are many models for example for communication and synchronization between processes developed in other fields, and although we believe that they are not directly applicable to data-driven coordination, they provide many important design suggestions for a formal D3C model.

Meeting this challenge requires an in-depth understanding of the range of data-driven coordination tasks that users may want to perform. The model must be expressive enough to be useful in a wide range of settings; for example, it must be able to express coordination constraints beyond the simple "I want to sit next to my friend on the plane" variety. At the minimum, the abstraction must be able to express global constraints such as "No two meetings may be booked in the same time slot", or "Seats on a plane that is not full must be allocated in a way to keep it weight-balanced". Further, in auctions, donations, or other financial settings, numerical constraints such as the following are needed: "I will match all other club members' donations up to x dollars".

Programming Model. To be successful and useful, the coordination model needs more than just expressiveness. It

must be associated with a natural and intuitive programming paradigm, so that users can easily specify their coordination parameters. This has long been a challenge in the general field of concurrent programming, and it remains so for data-driven coordination. A declarative model will help in achieving clarity, as it moves complexity away from the programmer and into the system. This calls for a clean design of the actual high-level language constructs that programmers will interact with.

Queries on the Coordination State. Users may want to query the system about past and pending coordination events. For example, in a gaming scenario, a user may want to know if there are others who have expressed an intention to attack and formulate her own strategy based on this information. A user may also wish to pose such queries to understand why a coordination attempt has failed. We need to develop a suitable language for these coordination state queries that is easy to use and compatible with the coordination abstraction itself. Maybe it is even possible to represent past and pending coordinations in the system catalog and they thus become available through the standard SQL interface to the database system.

3.2 Achieving Efficient Coordination

A powerful abstraction and a declarative language for coordination are important, but neither of them will be useful without an efficient and scalable implementation. With the coordination model in place, the design of efficient algorithms to carry out the actual coordination in the system is the next challenge.

Expressiveness Versus Complexity. In the previous section, we stressed the need for the coordination model to be expressive. It is a given that with great (expressive) power comes great computational complexity. Work in some specific domains such as donation matching [2] shows clearly that the coordination problem becomes NP-complete rapidly as the expressiveness of the language increases. Therefore the development of the coordination abstraction should needs to consider practically useful tradeoff points between expressiveness and computational complexity.

Heuristics and Restrictions. In the presence of intractability, restricting expressiveness may not always be an acceptable solution. Therefore, addressing the efficiency challenge must include other strategies such as realistic restrictions on the problem instances. Such restrictions may, for example, have to do with properties of the specific workload (e.g., bounded treewidth of an underlying graph).

Efficient Algorithms. In addition to developing coordination algorithms of low computational complexity, we must search for ways to optimize them so that they scale to the large coordination settings that we imagine. There is wide scope to develop optimizations for coordination, particularly for settings with a very large number of users and coordination requests. For example, in many data-driven coordination scenarios, although the number of coordination requests in the system may be high, the requests are likely to be broadly similar (e.g., they may share query templates but differ in parameters). This will be the case, for example, with travel planning, course selection, and scheduling. Based on this observation, it may be worth exploring to what extent the coordination requests can be processed in batches rather than individually, perhaps using techniques similar to those designed for multiquery optimization [10, 3].

Another idea how to improve performance is through indexing and caching. In many scenarios (for example, in travel planning) users will not submit their coordination requests to the system simultaneously; we need a way to buffer the requests as they arrive and wait for a coordination partner. A suitable data structure can make it possible to quickly determine for an arriving coordination request whether a matching partner request exists in the system.

The feasibility of some of these optimizations will depend on where in the system coordination is implemented – i.e. whether it sits at the application layer or in the DBMS. We return to this issue in Section 3.5.

3.3 Integrating Coordination into Transactions

Concurrency control. Understanding and implementing coordination as a basic primitive is a fundamental step, but it is, however, only the first step. Generally, users expect to use coordination within larger transactions with atomicity and durability guarantees. Also, although coordination is by definition a breach of isolation, it is clear that whenever a transaction contains *non-coordinating* code, this code should be executed in isolation as far as possible. D3C must therefore be based on a concurrency control model with the power to specify exactly what to coordinate and what to isolate — and all of this without much complexity to the programmer.

Addressing this challenge involves a fundamental reassessment of the classical notion of isolation. Traditionally, isolation for transaction schedules has been defined in terms of serializability, that is, equivalence to a serial schedule. With coordinating transactions, serial execution of individual transactions is normally not possible: no single transaction may be able to proceed past a coordination step unless a partner is available.

Beyond issues of isolation, the concurrency control model needs to address issues such as the handling of dependencies which are created between transactions when they coordinate. For example, if two friends book tickets together and one of the transactions aborts, the other transaction may be required to abort as well for correctness.

Finally, once a suitable model for concurrency control is in place, it is a further challenge to figure out how to enforce it in practice via efficient protocols.

3.4 Balancing Coordination and Privacy

Coordination as an Opt-In Mechanism. While coordination inherently allows bidirectional information flow between transactions, care must be taken in designing the co-

ordination abstraction to ensure that coordination is an "opt-in" process. In many settings that we have discussed, users may want to coordinate selectively with other users and also may only want to coordinate particular aspects of their transactions. It is important that the abstraction supports such access control and prevents any unintentional information flow while still remaining flexible enough to promote coordination.

Privacy in Coordination State Queries. We have mentioned that allowing users to query the coordination state is potentially a very useful feature. However, such queries present a clear potential for privacy breaches as well; not everyone using the system may want to have their coordination requests visible to queries. Again, there is a need for a design that balances information flow and privacy; a first approach may be a careful consideration of access contol.

3.5 Architecting a System for D3C

The basic models, algorithms and protocols discussed so far are the conceptual foundation of any system that is able to support D3C. On top of this foundation, we need to build to a robust, scalable and – above all – *useful* system. This task requires a deep understanding of the architectural tradeoffs involved.

Deciding Where to Implement D3C. The most obvious tradeoff is whether the coordination should be performed within the database system itself, or outside at the application layer. Both choices come with significant advantages and disadvantages. Implementing coordination at the application layer may be simpler and easier to add to existing systems. On the other hand, as we have already discussed, the data-intensive coordination algorithms are poised to benefit significantly from deep optimizations which will be most successful if implemented at the database level.

Implications of D3C on the Architecture of a Database System. D3C as a paradigm demands a rethinking of many database fundamentals. Up until now, the need to maintain isolation among processes working on the same data has been a basic "given". As such, it permeates all aspects of the design of today's database systems for example the buffer manager, the lock manager, and query processing. When isolation is no longer guaranteed, we may benefit from re-thinking how to architect such a system. Understanding the impact on the design of a DBMS by moving from an isolation-only concurrency model for data processing to a model where both isolation and coordination are present is probably the most far-reaching research challenge of all.

4. D3C IN ACTION

In an attempt to get a handle on the challenges introduced in the previous section, we at Cornell are developing Youtopia, a prototype system that supports data-driven coordination. Here, we describe the proposed architecture of the system and a proof-of-concept application based on Youtopia that allows a user to coordinate travel plans with Facebook friends.

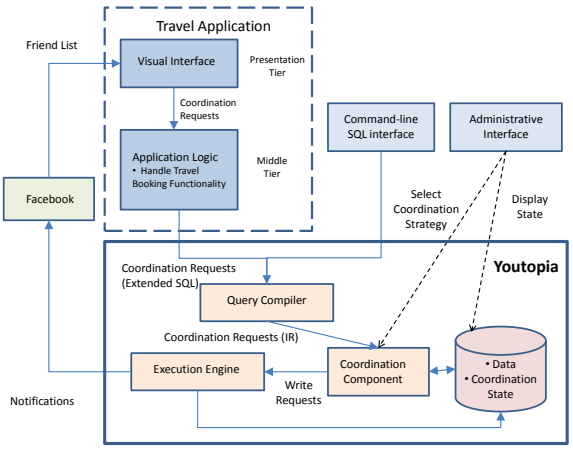

Figure 1: Draft Youtopia Architecture

Figure 2: Coordinating a Flight Booking With Friends

Figure 1 shows the proposed architecture of our system. The coordination is handled within the DBMS. Users specify which friends they want to coordinate with using a visual interface shown in Figure 2. Based on user input, formal coordination requests are generated by the middle tier and submitted to the DBMS. A query compiler processes the requests and translates them to an intermediate representation inside the DBMS for processing by the coordination component.

The coordination component runs whenever a relevant request arrives in the system, and it is able to run two different coordination algorithms. The choice of the best algorithm depends on the specific system workload; in our system this choice can be made through the administrative interface. The execution engine uses the output of the coordination algorithm and actually carries out the flight bookings to match the users' requests.

Youtopia will also provide an SQL command line interface which allows SQL and coordination requests to be input directly to the system by the user. There will also be an administrative interface which can be used to view the current state of the database tables and the coordination-related internal state information.

While our prototype is still in its early stages, our initial results are very encouraging. We have implemented the travel application on top of Youtopia, following a standard three-tier architecture. The graphical frontend runs in a browser; it is an interface to all the functionality provided by the middle tier. The middle tier contains application logic to handle the standard functionality of a travel Web site: searching for flights and hotels, selecting specific flights and hotels. The middle tier also contains code to create coordination requests based on the user's friend list, and it has access to a special Youtopia API that allows it to provide an "account view", where users can see pending or confirmed reservations.

5. CONCLUSIONS

We believe that not only people, but also database systems can be highly effective, and that it is time for them to mature beyond independence to interdependence. Data-driven coordination brings not only valuable functionality but has also set the table for a feast of challenging research problems. Let us coordinate to address these challenges!

Acknowledgments. This research was supported by the New York State Foundation for Science, Technology, and Innovation under Agreement C050061 and by the National Science Foundation under Grants 0627680 and 0725260. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsors.

6. REFERENCES

[1] zope.stackless.com.

[2] V. Conitzer and T. Sandholm. Expressive negotiation over donations to charities. In *ACM Conference on Electronic Commerce*, pages 51–60, 2004.

[3] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, pages 761–772, 2007.

[4] S. L. P. Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL*, pages 295–308, 1996.

[5] G. Koutrika, B. Bercovitz, R. Ikeda, F. Kaliszan, H. Liou, Z. M. Zadeh, and H. Garcia-Molina. Social systems: Can we do more than just poke friends? In *CIDR*, 2009.

[6] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

[7] N. A. Lynch and M. Merritt. Introduction to the theory of nested transactions. *Theor. Comput. Sci.*, 62(1-2):123, 1988.

[8] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[9] J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[10] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, 2000.

[11] R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[12] G. Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, 1999.

[13] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.