All you need is ASK: Leveraging Application-Specific Knowledge to Build Databases

Cong Yan, Guna Prasaad, Alvin Cheung, Dan Suciu (University of Washington)

ABSTRACT

Current query interfaces exposed by database systems are too limiting for query optimization, and database systems have to resort to gather additional information from side channels such as query logs or maintain data statistics for such task. In this paper, we present a new language called ASK for developers to provide application-specific information to the database system. To demonstrate the power of ASK, we have used the gathered information to optimize the query planner and the transaction scheduler in the ANSWER database system. Using standard benchmarks and two real-world database-backed applications, we show that ANSWER can improve the resulting application query execution by up to 145×, and transaction throughput by up to 5×, as compared to an implementation that is agnostic to such knowledge.

1 INTRODUCTION

Since their initial development, all database management systems (DBMSs) interact with their users via a "REPL[1]-like" interface: users issue queries written in a query language such as SQL, the DBMS evaluates the query, and returns results. Such interface is well-suited for interactive applications, such as those where users issue queries directly from the terminal, or where the exact query to be issued can be embedded in the application. While simple, this interface results in the DBMSs having only extremely limited knowledge about why queries are issued, or the patterns of queries that are issued by the application. As such, DBMSs can only resort to using "side channels," like data histograms and query logs, to infer user intentions, and needing to guess such information makes query optimization difficult.

The development of libraries (e.g., JDBC), along with high-level languages (e.g., LINQ) and object-relational mapping (ORM) frameworks (e.g., Rails, Hibernate) in the last decade have provided an alternate way to issue queries: rather than typing free-form text or issuing exact canned queries through the REPL interface, users now issue queries by interacting with the application instead, and the application in turn uses such libraries and frameworks to translate their data needs into queries. As a result, many queries are now *programatically composed* by assembling various code templates that are embedded within applications together with user inputs, for instance by calling functions or following different control flow paths in the application based on user interaction.

Yet, little has been done to exploit this new query issuing mechanism in query optimization or the construction of DBMSs in general. In fact, recent studies [17] have shown that, without a detailed understanding of the internals of DBMSs, application developers using such interfaces to issue queries often result in misusing them despite their attempts to convey application intention by writing stylized (and sometimes awkward looking) queries, and this leads to both significant application performance degradation and software maintenance challenges [3, 16].

We believe that there is a wealth of *Application-Specific Knowledge* (i.e., ASK) that developers can easily (and are often willing to!) provide to the DBMS to aid in query execution. Unfortunately, this information is very difficult to convey using the current interfaces provided by the DBMSs. To that end, we propose ASK, an API for application developers to express high-level semantic information about their applications to the DBMS. ASK is designed to complement the existing query-issuing interfaces that DBMSs provide by allowing application developers to convey additional information about their application, such as data access patterns and logical properties of the stored data, all of which are difficult to infer from query logs, histograms, or analysis of application code.

To illustrate the effectiveness of ASK, we propose ANSWER, a new in-memory multi-core transactional DBMS we are building that leverages the information conveyed via ASK. We have constructed two major components in ANSWER thus far: the query planner and the transaction scheduler. While the basic design of the two components resemble typical implementations, by incorporating application-specific information, ANSWER offers substantially better application performance (up to 145× speedup in query execution and 2× in transaction throughput) as compared to a generic implementation that does not exploit such knowledge, as demonstrated using two standard database benchmarks and two real-world applications.

2 AN API FOR APPLICATION-SPECIFIC KNOWLEDGE

In this section we describe ASK, our API for developers to convey application-specific knowledge to the DBMS. As queries are often assembled in a piecewise manner from different

[1]Read-Eval-Print Loop

control flow paths (see Section 3.3 for examples), automatically inferring the structure of queries from application code is difficult. Doing so from the query logs is not easy either, as the generated queries often differ due to user behavior, despite them being issued from the same location (e.g., one user might prefer sorting news items based on time while another might prefer relevance, even though the code that generates the query retrieve the news items stem from the same location). However, developers might be aware of query fragments that should be present in the generated query regardless of the different control paths that the program might take. Hance, ASK provides two functions for developers to convey such application-specific knowledge:

- `template(fragment)` is used to describe query fragments that are present in the generated query. Templates can either be fully specified, e.g., `q.template("status=0")` means that `q`'s contains the predicate `status=0`. Templates can be partially specified as well, e.g., `q.template("{cost,quantity} > ?")` means that `q` contains a predicate where the attribute to be compared can be one of those in the provided list, depending on the program path taken, and the comparison value being a user-provided runtime parameter. A more general form, `q.template("?? > ?")`, would leave the comparison attribute unspecified as well.
- `path(c1, c2, ...)` specifies retrievals of `c2` objects frequently originate from a `c1` object. For example, `path(Project, Issue)` indicates `Issues` are accessed only from `Projects`, such as from the project that the issue is associated with.

Finally, the ASK API includes `conflictFree(f)` to convey the logical properties that are inherent in the data. The developer supplies a function `f` to `conflictFree`, where `f` that takes in two transactions as parameters and return true if they do not access the same data items. As we will discuss in Section 4, such information is useful for scheduling transactions but is difficult to determine from query logs or the application code.

The ASK API is currently designed to be used by developers constructing applications using various ORM frameworks. To use ASK, developers simply insert calls to the ASK functions in their application code. Using ASK, we are currently implementing ANSWER, a new in-memory transactional DBMS. We next describe the two ANSWER components that we have built, the query planner and the transaction scheduler, and how we leverage application-specific information in those two components to improve application performance.

3 BUILDING AN APPLICATION-SPECIFIC QUERY PLANNER

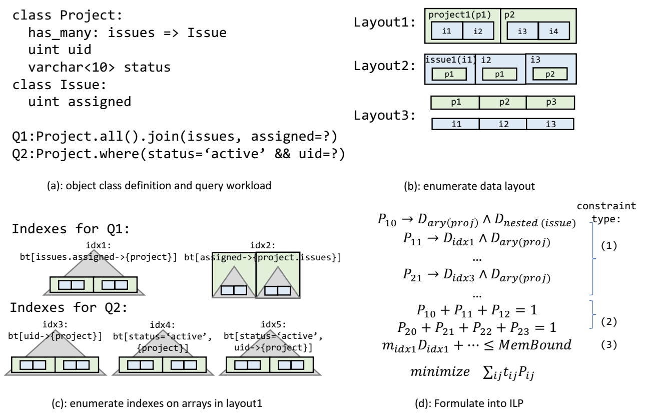
In this section we describe how the ANSWER query planner leverages the application-specific information conveyed by developers using ASK to improve performance.

3.1 Query planner design

The goal of ANSWER's query planner is to find the best physical design (i.e., the data structures for holding the data in memory along with indexes) and query plans that minimizes the query time while not exceeding available memory. It takes as input an ASK-annotated application source code built using an ORM framework (such as the Rails API [7]) and a user-provided parameter that specifies the amount of memory available for the database.

ANSWER's query planner has a simple design that is inspired by relational physical designers [1], except that it searches for data layouts in addition to indexes. A data layout describes how objects are stored in memory. ANSWER current considers storing all objects of the same class consecutively, or nest them in another class (e.g., nest `Issues` within their corresponding `Projects`). For each data layout, the planner enumerates the indexes that can be utilized by the queries. The indexes together with the layout forms a physical design. For each physical design, ANSWER estimates the execution time of the corresponding query plan using a static cost model. We omit the cost model details due to space.

To find the optimal design, ANSWER's planner formulates the problem as an Integer Linear Programming (ILP) problem, with the constraints being that 1) each physical design is defined, 2) each query uses only one design, and 3) the total memory used by all queries is smaller than the user-provided bound. The ILP's goal is to minimize the sum of execution time of all queries. Solving it gives the optimal physical design for the each query, and simple code generation rules are then used to create executable query plans given the physical design.



```
class Project:
  has_many: issues => Issue
  uint uid
  varchar<10> status
class Issue:
  uint assigned

Q1:Project.all().join(issues, assigned=?)
Q2:Project.where(status=‘active’ && uid=?)
```

(a): object class definition and query workload
Layout1:
project1(p1)
p2
i1
i2
i3
i4
Layout2:
issue1(i1)
i2
i3
p1
p1
p2
Layout3:
p1
p2
p3
i1
i2
i3
(b): enumerate data layout
Indexes for Q1:
idx1:
bt[issues.assigned->{project}]
idx2:
bt[assigned->{project.issues}]
Indexes for Q2:
idx3:
bt[uid->{project}]
idx4:
bt[status=‘active’, {project}]
idx5:
bt[status=‘active’, uid->{project}]
(c): enumerate indexes on arrays in layout1
constraint type:
(d): Formulate into ILP

$$P_{10} \rightarrow D_{ary(proj)} \wedge D_{nested\ (issue)}$$
$$P_{11} \rightarrow D_{idx1} \wedge D_{ary(proj)}$$
$$\ldots$$
$$P_{21} \rightarrow D_{idx3} \wedge D_{ary(proj)}$$
$$\ldots \qquad (1)$$
$$P_{10} + P_{11} + P_{12} = 1$$
$$P_{20} + P_{21} + P_{22} + P_{23} = 1 \qquad (2)$$
$$m_{idx1} D_{idx1} + \cdots \leq MemBound \qquad (3)$$
$$minimize \quad \sum_{ij} t_{ij} P_{ij}$$

Figure 1: An example workflow of ANSWER's planner.

Figure 1 illustrates how ANSWER's planner works: (a) shows the Rails input source code (abridged from a project management application [6]) that contains class definitions and a workload of two queries. `Q1` is an inner join on `Projects` and `Issues` that are `assigned` to an individual (whose value is provided by the user at runtime). It returns a list of `Projects`, each containing a list of `Issues`. `Q2` is a simple selection. (b) then shows the three different data layouts that ANSWER's planner considers. In layout 1, `Issues` are stored as nested objects within `Projects` in memory (vice versa in layout 2), and in layout 3, both `Issues` and `Projects` are stored separately. (c) shows the indexes that ANSWER considers for `Q1` and `Q2`, where an index can be constructed on a subset of objects (we denote this as `index_type[data partition, key type -> value type]`). For instance, `bt[status='active', uid->project]` encodes a B-tree index created on only the active `Projects`, with key as user id, each mapping to a list of `Projects` owned by that user. (d) shows the ILP formulation where P_{ij} is the binary variable as indicator for j-th physical design of query i, and D_{idx1}, $D_{ary(proj)}$ are indicators for `idx1` (shown in (c)) and array of `Projects`, respectively.

3.2 Search space and search algorithm

As discussed, the search space considered by ANSWER includes different data layouts and indexes. ANSWER currently uses a simple enumeration approach: when enumerating data layouts, for every join (e.g., `Q1` in Figure 1(a)) in a query Q, the planner enumerates all layouts for the two classes involved in the join. To search for indexes for Q, the planner enumerates all combinations of attributes and data partitions.

While simple, this strategy results in a huge search space. ANSWER does not use heuristic to reduce the search space. Instead, it relies on application-specific information provided by developers using ASK to prune the search space.

3.3 Leveraging ASK for query planning

As queries are generated programmatically, similar query fragments can appear either within one query or across multiple queries. Developers can specify such fragments as templates as described in Section 2. ANSWER uses such information to prune the search space for indexes and data layouts. It processes the templates differently depending on whether each is fully or partially specified.

Fully-specified templates. As discussed in Section 2, a fully-specified template does not contain any placeholders to be determined during query construction. For queries annotated with such templates, ANSWER searches only for indexes that can be used to speed up execution of the query fragments in the templates, rather than searching for all indexes.

For example, Listing 1 lists two queries from a forum application. `q1` returns unexpired posts authored by a user provided value (`param["uid"]`) ordered by time, while `q2` returns unexpired posts with positive votes with a specific author. The presence of function calls (`authorOf`, `unexpired`, etc) in the code makes it difficult to automatically infer the resulting query structure. Using `template`, however, informs ANSWER that both `q1` and `q2` contain predicates on `expired` and `uid`. As an index on `uid` for the data partition `expired=false` can be used to speed up `q1` (and similarly for `q2`). Hence ANSWER's planner will only consider the indexes `bt[expired=false, (uid,created)->{post}]` and `bt[expired=false, (uid)->{post}]` for `q1` as driven by the template, and **not** consider indexes like `bt[uid->{post}]`, which is less useful. This reduces the number of indexes to be considered from 6 to 2.

Listing 1: Code example with a fully-specified template, with the final generated queries in comments

```
t = Template("expired=false && uid=?")
q1 = sortByTime(authorOf(unexpired(),
                          param["uid"])).template(t)
// q1 = SELECT * FROM Posts
//      WHERE expired=false && uid=? ORDER BY time
q2 = authorOf(hasVotes(unexpired()),
                        param["uid"])).template(t)
// q2 = SELECT * FROM Post
//      WHERE expired=false && uid=? && votes>0
```

Partially-specified templates. Similarly, ANSWER also leverages the partially-specified templates to prune the index search space. For example, when there are multiple parts in a query that uses one template, ANSWER only searches for indexes that can be utilized by the first part, and then uses it to answer other parts within the query. For example, in Listing 2, `q` is a union of two query fragments: the first returns `Members` belonging to `Projects` with more than a user-specified member size, and the second returns `Members` of `Groups` with a similar member size requirement. In this case, the indexes that can be used to answer these two fragments are similar, i.e., the same indexes or indexes on the same attribute(s), but constructed on different data partitions.

Listing 2: Code with a partially-specified template

```
t = Template("type=?? && source=?")
q = projectMembers(enoughMembers(param["sz"]))
    .union(groupMembers(enoughMembers(param["sz"])))
    .template(t)
// q = (SELECT * FROM Member
//     WHERE type='Project' && size>?) UNION ALL
//     (SELECT * FROM Member
//     WHERE type='Group' && size>?)
```

Like before, the presence of function calls obscures the structure of the final query. Hence, we define the template `t` and associate it with `q` in Listing 2. Given `t`, ANSWER matches each query fragment in `q` to this template, and only enumerates the indexes for the first fragment. For each index enumerated, e.g., `bt[type='Project', size->{member}]`, ANSWER assigns a similar index—an index that use the same (partial) predicate in the template, `bt[type='Member', size->{member}]`— to answer the second fragment. ANSWER will not consider using the index `bt[type='Project']` to answer the first fragment and `bt[size->{member}]` to answer the second. This reduces the search space of physical designs from 4^2 (a cross product of designs for the two fragments) to only 4.

Leveraging object access paths. ANSWER leverages the ASK path information to prune the search space for data layouts. For example, as shown in Figure 1, the annotation `path(Project, Issue)` informs ANSWER's planner to consider only layout 1, since other layouts are not helpful in improving the execution time of `Q1`. As a comparison, using layout 2 would require reshaping `Issues` that are with nested `Project` into `Project` objects with nested `Issues` instead. As shown, the ASK annotation allows ANSWER to reduce the number of layouts to consider from 3 to 1. In general, an application often have many classes that are stored persistently. Developers can provide hints regarding the access patterns of a subset of the classes in the application, and rely on ANSWER to search for the optimal layouts for the rest.

3.4 Evaluation

We have implemented a prototype query planner in ANSWER and evaluated it using two open-source applications [5] [6] and the TPC-H benchmark [4]. We picked 32 queries for Lobsters, 26 queries from Redmine and 8 from TPC-H (queries 1, 3-6, 12, 14, and 19) as workload. We evaluated the number of physical designs considered by ANSWER, and the search time for the optimal physical design under four settings: 1) without using ASK (denoted as "Orig"), 2) using ASK to specify only query templates ("Template"), 3) specifying only layout ("Layout"), and 4) specifying both template and layout ("Both"). The run time includes both the search of physical designs and ILP solving time. Finally, we compare the resulting query performance of ANSWER with MySQL where we have added the indexes that would have been created by AutoAdmin [2] given the workload.

The result is shown in Table 1. Originally the search space of ANSWER is very large, containing hundreds of thousand of physical designs, and the search process is slow, spending up to 312 minutes even for a small workload. With only a few specified templates and layouts using ASK, however, the search time is reduced to only a few minutes. The ANSWER chosen physical designs also greatly improve the query performance comparing to MySQL, from 1.5× to 145×.

4 UTILIZING ASK FOR TRANSACTION PROCESSING

In this section we describe how ANSWER leverage the data property expressed using the `conflictFree` API call from ASK with a new transaction scheduling algorithm.

4.1 Transaction processing in ANSWER

Transactions in ANSWER are executed in batches. Incoming transactions are collected into batches, and each batch is executed to completion before processing the next one. The key insight behind the ANSWER transaction scheduler is that when two transactions have conflicting access (with at least one being a write) to the same data item, we can eliminate concurrency control and still guarantee serializability by scheduling both transactions to be executed on the same core. On the other hand, if two transactions access completely disjoint data items, they can be scheduled on different cores to exploit parallelism.

Realizing this insight requires splitting each incoming batch of transactions into *conflict-free* clusters where there are no data access conflicts between any two clusters (and can thus be executed across different cores without concurrency control), and a *residual* cluster containing the rest of the transactions to be executed with concurrency control. To do so, ANSWER's scheduler first creates an access graph for each batch of transactions: $G = (D, B, E)$, a bi-partite graph between the data items D and transactions B where a transaction is connected to each data item it accesses. The scheduler then proceeds in two steps:

- **Spot**: The scheduler first randomly samples a fixed number of transactions from G that serve as *initial seed transactions* for the conflict-free clusters. If a chosen transaction T has a data conflict with another seed transaction, T is discarded and another transaction is sampled.
- **Allocate**: Given the seed clusters, the scheduler then scan through the rest of the transactions and assign each to a seed cluster. A transaction T is assigned to a cluster C if T accesses data items that are accessed by at least one other transaction assigned to C. If T accesses data items that span across two clusters, then we either merge the two clusters, or assign T to the residual cluster based on a user provided parameter. For transaction T' that is not assigned to a cluster at the end of **Allocate** (as no other transactions share the same data item accesses as T'), the scheduler then assigns it randomly to one of the existing clusters and run **Allocate** again to schedule the remaining unallocated transactions, if any.

After this process, ANSWER's scheduler executes the transactions in conflict-free clusters without concurrency control across different cores. Once all such clusters are executed completely, it then executes the transactions in the residual cluster with concurrency control on all cores as in a traditional transaction processing system.

Figure 2 demonstrates ANSWER's scheduling algorithm using a workload of three transactions (t_1 to t_3) and four data items (d_1 to d_4). In (a), only t_1 is sampled during the **Spot** phase. This results in a single conflict-free cluster with t_2 and

	Lobsters [5]				Redmine [6]				TPC-H			
	Orig	Template	Layout	Both	Orig	Template	Layout	Both	Orig	Template	Layout	Both
#ASK-API calls	0	4	3	7	0	3	5	8	0	1	5	6
#designs	645K	143K	147K	72K	268K	66K	54K	14K	529K	23K	461K	5K
planner time	233min	47min	42min	18min	91min	18min	14min	3min	312min	12min	287min	2min
Avg query	MySQL index		ANSWER		MySQL index		ANSWER		MySQL index		ANSWER	
execution time	0.45sec		0.3sec		1.7sec		0.11sec		91.60sec		0.63sec	

Table 1: Search space and running time of ANSWER' query planner and query performance

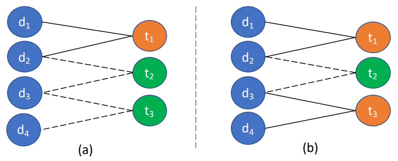


Figure 2: Illustration of ANSWER's scheduling algorithm using a bi-partite graph on data items and transactions. Lines represent data accesses. Orange transactions are the cluster seeds, and green transactions are assigned during the Allocate phase.

t_3 both assigned to it, and the cluster is executed on a single core without any concurrency control. In contrast, in (b) both t_1 and t_3 are sampled, and two seed clusters are created. During **Allocate**, t_2 either causes the two clusters to merge and revert to the same case as (a) (after wasting the initial sampling efforts in considering t_3 as its own seed cluster), or is assigned to the residuals and executed with concurrency control. Both scenarios are suboptimal as compared to (a).

4.2 Improving scheduling using ASK

The performance of ANSWER's transaction processing component relies on the number of transactions that can be assigned to the conflict-free clusters, as they are executed concurrently without concurrency control. As an extreme, while assigning all transactions to the residual preserves serializability, it will degrade the performance of ANSWER substantially. This depends critically on the initial sampling in finding "good" cluster seeds.

To illustrate how ANSWER leverages application-specific knowledge for transaction scheduling, consider the TPC-C benchmark, where two payment (or new order) transactions can be executed in parallel without concurrency control if they access items from different warehouses. We capture this application-specific knowledge using ASK as shown in Listing 3, with the function passed to `conflictFree` returning `false` if `t1` and `t2` access items from different warehouses.

Listing 3: ASK specification for TPC-C

```
conflictFree((t1, t2) -> {
  wh1 = warehouses(t1) // warehouses for t1's items
  wh2 = warehouses(t2) // warehouses for t2's items
  return (wh1.intersect(wh2) == null) })
```

ANSWER leverages the user-provided `conflictFree` annotations by using it to identify the cluster seeds during sampling. Consider two transactions, T_1 and T_2, that read items from the same warehouse. Without the user-provided ASK annotation, it is possible that both transactions are sampled and used as clusters seeds, similar to Figure 2. If the resulting clusters are not merged, then any other transaction T' that accesses items from both clusters might be allocated to the residuals and executed with concurrency control, while a better execution scheme is to assign all three to the same core and execute them without concurrency control. With the annotation, however, `conflictFree` returns `false` for T_1 and T_2, and hence ANSWER will not use both as cluster seeds. This greatly improves the resulting quality of the conflict-free clusters and system throughput as our experiments show.

4.3 Evaluation

We have implemented the transaction scheduler in ANSWER and compared with several variants of 2PL and OCC [8]. We use two standard benchmarks in our experiments:

- **TPC-C**: We use a 50:50 mix of payment and new-order transactions (10 items/order), with 15% payments are made to a remote warehouse and 1% of items are ordered from a remote warehouse, resulting in 10% new-order transactions with at least one remote order. We use the `conflictFree` annotation as shown in Listing 3.
- **YCSB**: We simulate a marketplace that contains 15 categories of products derived from the YCSB workload. Each transaction orders 20 different items from a single category and the items are chosen using an intra-category Zipfian distribution. We experiment with 3 different values of the Zipfian constant to analyze the performance and use a `conflictFree` annotation that returns true for two transactions if they order products from different categories, similar to that shown in Listing 3.

The result of our TPC-C experiment on 15 cores is shown in Figure 3(a). We compared the throughput of ANSWER using 4 and 15 warehouses to model different amounts of contention. Since each transaction accesses the warehouse tuple, ANSWER generates 4 and 15 conflict-free clusters respectively by leveraging the user-provided ASK annotation. Meanwhile, all other protocols suffer as the highly contended warehouse tuples result in blocking (for lock-based protocols) or aborts (for OCC). In comparison, ANSWER outperforms all protocols by increasing throughput by almost 2× in both cases.

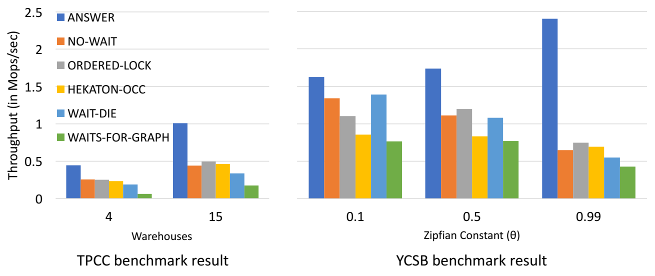


Figure 3: TPC-C and YCSB benchmark results

The YCSB experiment on 15 cores is shown in Figure 3(b) for 15 categories and various values of the Zipfian constant θ. The Zipfian constant controls the contention within each category, with 0.1, 0.5, and 0.99 represent low, medium and high contention respectively. ANSWER performs better than other protocols in all these cases by leveraging application-specific information and creating the optimal number of conflict-free clusters across the different contention scenarios, with almost 5× better compared to other protocols for the high contention (θ=0.99) case by utilizing application-specific knowledge provided by the developer.

5 RELATED WORK

Automated physical designers. Much work has been done on automating physical designers, with the pioneering Autoadmin [1, 2] tool using workload information from query logs to suggest the best indexes and materialized views. It uses heuristics to reduce search space, e.g., by considering only indexes up to a pre-determined number of fields. In contrast, ANSWER does not use heuristics. Instead it relies on the application-specific knowledge provided by developer to reduce the search space. Our evaluation shows that leveraging such knowledge directly is more effective in searching for the optimal design.

Caching intermediate query results. ANSWER's leverage of ASK templates resembles prior work on using common subexpressions to speed up multi-query optimization [13] and plan sharing across multiple queries [9]. However, prior work relies on complex analysis to automatically infer such subexpressions and shareable query plans, while ANSWER solicits such knowledge directly from developers.

Concurrency control protocols. Various protocols have been proposed to reduce the amount of overhead associated with executing the protocol [10, 11], and also adaptively changing which protocol to apply based on workload [14]. Meanwhile, new systems have been constructed to eliminate concurrency control completely [12, 15]. Using such systems require entire applications to be rewritten. ANSWER does not have such requirement. Instead it relies on developers providing knowledge about their applications, and utilizes such knowledge to schedule transactions.

6 CONCLUSION

In this paper we discussed ASK, a new language for developers to convey application-specific information to the DBMS. We have demonstrated that, by leveraging such information, we can improve both query execution performance by up to 145×, and transaction throughput of the resulting application by up to 5×, using both the TPC-C and TPC-H benchmarks, along with two real-world database-backed applications.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*. 496–505.
[2] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.
[3] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping. In *ICSE*. 1001–1012.
[4] Transaction Processing Performance Council. 1999. TPC-H benchmark. http://www.tpc.org/information/benchmarks.asp.
[5] Lobsters developers. 2018. Lobsters, a forum application. https://github.com/lobsters/lobsters.
[6] Redmine developers. 2018. Redmine, a project management application. https://github.com/redmine/redmine.
[7] Rails developers. 2018. Ruby on Rails query API. https://guides.rubyonrails.org/active_record_querying.html.
[8] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*. 1243–1254.
[9] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries With One Stone. *PVLDB* (2012), 526–537.
[10] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. 2009. Improving OLTP Scalability Using Speculative Lock Inheritance. *Proc. VLDB Endow.* (2009), 479–489.
[11] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. 2013. A Scalable Lock Manager for Multicores. In *SIGMOD*. 73–84.
[12] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* (2008), 1496–1499.
[13] Timos K. Sellis. 1988. Multiple-query Optimization. *ACM Trans. Database Syst.* (1988), 23–52.
[14] Dixin Tang and Aaron J. Elmore. 2018. Toward Coordination-free and Reconfigurable Mixed Concurrency Control. In *USENIX*. 809–822.
[15] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*. 1–12.
[16] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *CIKM*.
[17] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. How Not to Structure Your Database-backed Web Applications: A Study of Performance Bugs in the Wild. In *ICSE*. 800–810.