Generating Application-Specific Data Layouts for In-memory Databases

Cong Yan
University of Washington
congy@cs.washington.edu

Alvin Cheung
UC Berkeley *
akcheung@cs.berkeley.edu

ABSTRACT

Database applications are often developed with object-oriented languages while using relational databases as the backend. To accelerate these applications, developers would manually design customized data structures to store data in memory, and ways to utilize such data structures to answer queries. Doing so is brittle and requires a lot of effort. Alternatively, developers might automate the process by using relational physical design tools to create materialized views and indexes instead. However, the characteristics of object-oriented database applications are often distinct enough from traditional database applications such that classical relational query optimization techniques often cannot speed up queries that arise from such applications, as our experiments show.

To address this, we build CHESTNUT, a data layout generator for in-memory object-oriented database applications. Given a memory budget, CHESTNUT generates *customized* in-memory data layouts and query plans to answer queries written using a subset of the Rails API, a common framework for building object-oriented database applications. CHESTNUT differs from traditional query optimizers and physical designers in two ways. First, CHESTNUT automatically generates data layouts that are customized for the application after analyzing their queries, hence CHESTNUT-generated data layouts are designed to be efficient to answer queries from such applications. Second, CHESTNUT uses a novel enumeration and verification-based algorithm to generate query plans that use such data layouts, rather than rule-based approaches as in traditional query optimizers. We evaluated CHESTNUT on four open-source Rails database applications. The result shows that it can reduce average query processing time by over 3.6× (and up to 42×), as compared to other in-memory relational database engines.

PVLDB Reference Format:
Cong Yan, Alvin Cheung. Generating Application-specific Data Layouts for In-memory Databases. *PVLDB*, 12(11): 1513-1525, 2019.
DOI: https://doi.org/10.14778/3342263.3342630

1. INTRODUCTION

Rather than directly embedding SQL queries into application code, database applications are increasingly written using object-oriented programming languages (such as Java, Python, or Ruby) that rely on different means to translate data retrieval operations into SQL, for instance object-relational mapping (ORM) frameworks such as Hibernate [10], Django [8], or Rails [18]. The object-oriented programming paradigm makes it easy to develop database applications: data to be persistently managed are organized into classes, each class is mapped to a relation, and each object instance becomes a tuple. Operations involving persistent data in the application are then converted into queries. For instance, a project management application would organize its data into `Project` objects with a member field that stores each project's corresponding `Issue` objects. `Projects` and `Issues` are stored in separate relations, with the association between them maintained using foreign key `project_id` stored in each `Issue`. Retrieving a `Project` with its associated `Issues` then becomes a relational join across the `project` and `issue` relations. The idea is that by utilizing relational databases, such object-oriented database applications (OODAs) can leverage relational query optimization techniques to become efficient.

In practice, however, OODAs often exhibit characteristics that make them distinct from traditional transactional processing or analytical applications. In particular:

- **Nested data model.** OODAs often come with objects containing fields of variable-length lists, making data model highly non-relational. For instance, a project containing a list of associated issues, and each issue containing a list of developers assigned to it, etc. While variable-length lists are common in classical database applications (e.g., storing the list of ordered items in the TPC-C benchmark), OODAs, being object-oriented, makes it very easy to create deep object hierarchies including circular ones. While this has been pointed out in a previous benchmark [29], modern object-oriented languages and frameworks have exacerbated the problem: the object hierarchy can easily reach more than 10 levels, with nearly half of the queries returning objects from multiple levels.
- **Many-way materialized joins.** Due to deep object hierarchies, simple object queries like retrieving `Projects` that contain issues can turn into a long chain of multi-way joins when translated into relational queries. Unlike analytical applications where such joins are also common, the join results are directly returned in OODAs rather than aggregated. As an example, a single query can involve as many as 6 joins in the OODAs used in evaluation, and return as much as 5GB of data. This makes the data structures used to store persistent data of crucial importance as standard row or column stores are not the best fit.
- **Serialization cost.** As the database and application represent data in different formats, moving data across them incurs serialization cost. This cost is pronounced in OODAs as queries often return a long list of hierarchical objects which requires converting materialized join result into objects and nested objects. Serialization easily takes longer than retrieving data, as our experiments show.

*Work done while at the University of Washington.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342630

• **Complex predicates.** OODA queries include many complex predicates, as many frameworks expose methods to filter collections of persistent objects that can be easily chained. For instance, `Project.where(create>'10/30').where(create<'12/1')` returns all `Projects` created in November. With each `where` call translated into a selection predicate, the final query often contains many (potentially overlapping or redundant) predicates as a result of passing a collection through different method calls. In fact, in our OODA evaluation corpus a single query can involve as many as 40 comparison predicates.

• **Simple writes.** Similar to transactional applications, most writes in OODAs touch very few number of objects (one object for each write query in our evaluation corpus). This makes write optimization (e.g., batch updates) a secondary concern.

The above aspects make OODAs challenging to optimize using standard query processing techniques. In this paper, we present CHESTNUT, an in-memory data layout[1] and query plan generator for OODAs. CHESTNUT leverages recent advances in program analysis, symbolic execution, and solvers to automatically *generate* a custom data layout given application code. To use CHESTNUT, user provides as input OODA source code (written using a subset of the Rails [18] API) and a memory budget. CHESTNUT then searches for the best in-memory data layout that optimizes for overall query performance, and outputs C++ code that loads data from disk, executes queries using the data layout, and updates in-memory data and disk storage for writes.[2]

Internally, CHESTNUT breaks down the data layout generation problem into three parts: for each query it first enumerates different representations for the stored objects (e.g., as standalone objects, pointers, or nested in other objects) and the data structures (indexes or arrays), and then enumerates query plans that use these data structures. The designs for different queries are then combined to find the globally optimal one for the entire application. Unlike traditional physical designers that only focus on index designs, CHESTNUT considers tabular and nested layout to store the data along with indexes, as well as customized query plans that utilize the generated data layout. To make enumeration and search tractable, CHESTNUT analyzes application code to determine its query needs and creates a custom search space given the application code. To search efficiently, CHESTNUT relies on two carefully designed internal representations (IRs) to succinctly describe the space of data layouts, and formulates the problem of finding the best model as an Integer Linear Programming (ILP) problem. Finally, CHESTNUT leverages symbolic execution and verification to determine how to use the optimal design to answer queries and generates executable code.

We have implemented a prototype of CHESTNUT, and used it to compare with three state-of-the-art in-memory databases using real-world OODAs. To our best knowledge, CHESTNUT is the first optimizer for OODAs that searches for all three aspects of query execution (data model, physical design, and execution) concurrently. In sum, this paper makes the following contributions:

• We design an engine that finds the best data layout for OODAs. Our engine does not rely on rule-based translation as classical optimizers. Instead, it analyzes the application code to generate a custom search space for the storage model, and uses a novel enumeration and verification-based algorithm to generate query plans.

• Given the search space, we formulate the search for the best data layout as an ILP problem. We also devise optimization techniques to reduce the problem size such that the ILP can be solved efficiently, with an average of 3 minutes of solving time when deployed on real-world applications (Section 8).

• We built a prototype of CHESTNUT and evaluated it on four popular, open-source OODAs built with Ruby on Rails framework. Compared to the original deployment using an in-memory version of MySQL, CHESTNUT improves overall application performance by up to 42× (with an average of 6×). It also outperforms popular in-memory databases with an average speedup of 3.9× and 3.6× respectively. CHESTNUT compares favorably even after additional indexes and materialized views are deployed (Section 8).

In the following, we first discuss related work and how is CHESTNUT different from previous tools (Section 2). Next, we give an overview of CHESTNUT (Section 3) and introduce the search space of data layout and query plan (Section 4). Then we introduce CHESTNUT's algorithm to enumerate query plans (Section 5 and Section 6) and ILP formulation (Section 7). We conclude by presenting experiment results on four web applications in Section 8.

2. RELATED WORK

Data layout design. There is much prior work on automatic data layout design. One line of work explores ways to store data rather than the traditional row or column-major stores for each single table. For instance, Hyrise [41] and H2O [26] store columns together as column groups instead of individually. Widetable [47] uses a denormalized schema to accelerate OLAP queries. ReCache [28] uses dynamic caching to store data in tabular and nested layout to accommodate heterogeneous data source like CSV and JSON data. While prior work focuses on a restricted set of layouts, CHESTNUT integrally explores tabular and nested data layout together with auxiliary indexes, and also how objects are nested within each other, which field and subset of objects to store, etc. Such aspects are important for OODAs as our experiments show. CHESTNUT also synthesizes query plans from the generated data layout instead of relying on standard relational query optimizer. Doing so allows CHESTNUT's plans to better utilize the new data layout.

Another line of work focuses on learning the best data structure. For instance, Idreos et al. [42] design a key-value store that learns the configuration parameters to each or combinations of elementary data structures given query patterns. While reducing query execution time, such work focuses on queries with simple patterns on a single table. Queries in OODAs, which CHESTNUT optimizes, have complex query patterns involving many classes, and that leads to new challenges in deciding how to store objects of different classes and picking the optimal design across multiple queries.

Another approach is to use program synthesis [49, 48] to generate data structures from high-level specifications. While prior work optimizes only a single query without trading off update and memory, CHESTNUT handles workloads with many read and write queries subject to a memory bound and also determines how to share data structures among queries.

Physical design. Automated physical design for relational databases is a well-studied topic. AutoAdmin [24, 32, 25, 31] finds best indexes and materialized views by trying different possibilities and asking the query optimizer for the cost. To reduce the number of optimizer calls, it uses heuristics to consider only a small number of candidates. In contrast, CHESTNUT employs a different approach that opens the "black box" of the query optimizer and uses program synthesis to enumerate query plans instead of "what-if" calls.

CHESTNUT is not the only work that leverages ILP solver for physical design. Cophy [37] and CORADD [45] use ILP solvers to

[1] In this paper we use the term "data layout" to refer to both the data being stored (in cases where only a subset of fields in an object and a subset of all objects are stored), and the type of data structure used to store data in memory.

[2] CHESTNUT currently focuses on data layout design and considers other database features like concurrency control as future work.

find the best indexes and materialized views to add within a memory bound. NoSE [51] uses ILP to discover column families for NoSQL queries. Bruno et al. [30] use ILP to find the best sort order and pre-joined views for column store, and BIGSUBS [43] uses ILP to select common subexpressions that benefit the workload the most if materialized. These projects show that ILP is an attractive way to solve physical design problems. However, as CHESTNUT searches not just over relational (1NF) designs but also non-relational ones like nested objects and partial indexes, its formulation is different from prior work due to the nature of the problem.

Machine learning models are also used for physical design. Prior work [50, 52, 46] uses reinforcement learning to decide on join order. They reuse the query optimizer and replaces heuristics with machine learning models to utilize historical query performance to fine-tune plan choices. CHESTNUT replaces heuristics with efficient plan enumeration of the search space. It can find better plans as optimizers often do not have comprehensive rewriting rules.

Object-oriented and document databases. CHESTNUT leverages data structure concepts from object-oriented database systems [27, 44], such as the nested object model. Previous physical designers for OODBs focus on devising a language to describe the physical design [40, 38] or a single storage design that aims to optimize all OODAs [3]. Instead, CHESTNUT tailors the storage design to each application. CHESTNUT also proposes a language for storing objects and for query plan, but uses it for verification purposes.

Other NoSQL stores like document databases also store data in non-tabular format. However, the data model of document-oriented DBMS is different from OODAs. Document database's data model fixes the way how data are nested as defined by its JSON schema. Besides, document databases often support only limited query APIs, unlike Rails. These aspects make document databases unsuitable for OODAs given the complexity of such applications' query needs.

Program analysis for database applications. Previous research has used program analysis to help improve database applications, especially database applications that embed SQL queries in object-oriented programs. QURO [56] reorders queries without changing application semantics, Powerstation [59] replaces inefficient ORM API uses, QBS [34] optimizes query compilation by converting functionalities written in object-oriented languages into database queries, and StatusQuo [33] moves functionalities between the application and the database. In contrast, CHESTNUT takes the queries already compiled by Rails as input and finds the best data layout. We believe techniques from prior work can be combined with CHESTNUT to further improve the performance of OODAs.

3. OVERVIEW

In this section we give an overview of CHESTNUT using an example abridged from a project management application Redmine. This application [17] includes two classes to be managed persistently, `Project` and `Issue`, with a *one-to-many association* from project to issues, i.e., one `Project` can contain multiple `Issue` instances, but each `Issue` belongs to at most one `Project`. Listing 1 shows the definition of two classes and two queries using Rails API [18] supported by CHESTNUT: `Q1` returns `Projects` created later than a parameter `p1` that contain open issues,[3] and `Q2` updates an issue with `id` equals to `p2` and changes its status to `'close'`.

```
class Project:
  has_many: issues=>Issue # issues is a list of Issue
      objects, can be retrieved as a field of Project
  uint id
  date created
class Issue:
  has_one: project=>Project
  uint id
  string status
Q1 = Project.where(created>? AND exists(issues,status='
     open'), p1)
Q2 = Issue.update(p2, status:'close')
```

Listing 1: An example application

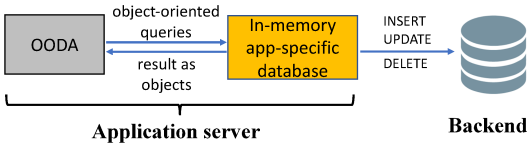


Figure 1: Using CHESTNUT's data layout for OODAs.

To use CHESTNUT, developer provides the input OODA code and a memory budget. CHESTNUT then generates the implementation of an in-memory data layout, consisting of data structures to hold data in memory, indexes to speed up queries, and code that leverages the data structures and indexes to answer queries embedded in the OODA. The total size of the data structures and indexes is subject to the provided memory budget. Figure 1 shows how the OODA can use the CHESTNUT-generated database. Like other in-memory databases, data is first loaded from persistent storage to in-memory data layout created by CHESTNUT. Embedded queries are implemented using CHESTNUT-generated code, with the results from writes propagated to persistent storage, and read results returned back to the application as in standard relational databases.

Internally, CHESTNUT searches for different ways to organize in-memory data given the application code. Figure 2 shows a few data layouts that CHESTNUT considers for Q1. In (a), the `Projects` and `Issues` are stored in two separate arrays, as in standard relational databases. Given this layout, to answer Q1, we scan the `Projects` array, and repeatedly scan the `Issues` array for each `Project` to find the associated open issues. Indexes can be used to avoid repeated scans. In (b), a B-tree index is created on the `created` field of `Project` to quickly find those newly-created `Projects`, while an index on the foreign key `project_id` of `Issue` can be used to find `Issues` associated with a `Project`.

We test the data layout above with 400K projects and 8M issues (altogether 3.2GB of data). Q1 takes 15s with 3.2GB of memory, and 9s with 3.6GB of memory to finish using (a) and (b), respectively. We can do better, however. Figure 2 (c) and (d) show another design that CHESTNUT considers, where (c) stores a nested array of `Issues` within each `Project` object, and the `Projects` are furthermore sorted on `created`. With (c), checking whether an open issue exists in a project can be done by checking the nested objects without scanning a separate `Issues` array. Using this layout, Q1 finishes in 7.4s with 3.2GB of memory. Q1 can run even faster with layout (d), which only stores `Projects` that contain open issues, and sorted by `created`. Using (d), Q1 requires only one range scan on the `Projects` array and finishes in 0.3s using only 0.2GB of memory. Note that as (c) stores nested objects, and (d) keeps only a subset of the `Projects` in memory as determined by Q1's predicate, neither of them would be considered by relational physical designers, but are within CHESTNUT's search space of designs.

We now briefly describe how CHESTNUT finds the best data layout. CHESTNUT first analyzes the OODA code and extracts all query componenets written with the query APIs. It then enumerates query plans for each individual read query, with each plan using a different data layout. To do so, it first enumerates object nestings, i.e., how associated objects are stored (such as `Projects` and the embedded `Issues` in Figure 2). It then generates data structures for each nest level that can be used to answer the query. These object

[3] "`exists`" is a short-hand expression introduced by CHESTNUT that filters projects based on their associated issues. The query can be written using standard Rails API as `Project.where("created>?",p).joins(issues,"status='open'").distinct`.

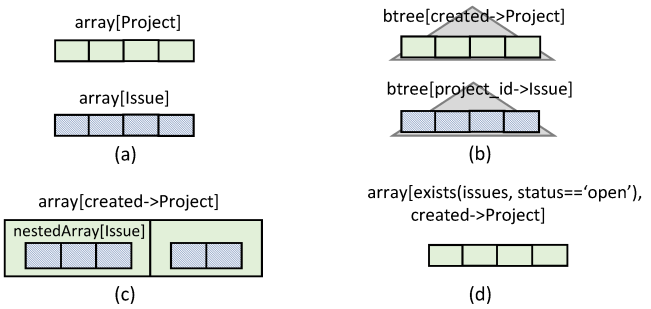


Figure 2: Example of data layouts for Q1. A green box shows a `Project` object and a blue shaded box shows an `Issue` object. A triangle shows an index created on the array of objects.

nestings and data structures define the search space of data layouts. To generate query plans, CHESTNUT enumerates over the space of query plans using the data structures. Such enumeration generates many invalid plans, and CHESTNUT verifies each plan against the query and retains only the valid ones. CHESTNUT also uses a few heuristics to prune out plans that are unlikely to be the best.

All data structures enumerated above can be potentially used to answer a read query, which need to be updated properly for writes. CHESTNUT generates plans for each write query on every data structure introduced when enumerating read plans. It first generates "reconnaissance" queries[4] that fetch the objects to be updated as well as other relevant data, and searches plans for these reconnaissance queries by treating them as normal read queries. During this process, new data structures may be introduced and they also need to be updated. CHESTNUT repeats the process until no new data structure is introduced and every seen data structure can be properly updated.

CHESTNUT formulates the problem of finding the data layout that optimizes *overall* OODA query performance as an ILP problem. Solving the ILP tells us which data structure to use for the data layout and the plan chosen for every query. CHESTNUT then generates C++ code that implements an in-memory application-specific database based on this result.

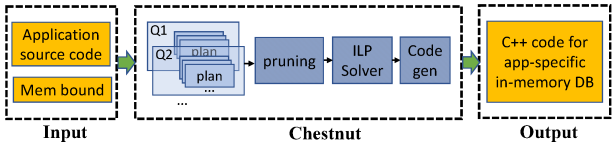


Figure 3: Workflow of CHESTNUT

The overall workflow of CHESTNUT is shown in Figure 3. We next describe each step in detail.

4. DATA LAYOUT AND QUERY PLAN

In this section we introduce the search space for data layouts and query plans.

4.1 Query Interface

CHESTNUT supports queries written using a subset of the Rails API [18], including commonly used query operations such as selection, projection, aggregation, etc [2]. As a review, all queries in Rails start with a class name (we refer to it as the "base class"), which by itself returns a list of all persistent objects of that class. Functions can be applied to the returned list, e.g., `where` and `find` along with a predicate, which is similar to relational selection. Join functions (e.g., `includes`, `joins`) are used in Rails queries to retrieve objects and their embedded (i.e., associated) fields, for instance, given the `has_many` association [1] between `Projects` and `Issues`, `Project.includes(:issues)` returns all projects that and their associated issues. Further predicates can be specified to filter the join result, for instance `Q1` shown in Listing 1 returns only projects with open issues using the CHESTNUT-provided `exists` shorthand.

When CHESTNUT analyzes the queries in the input OODA, it first rewrites them into a stylized form for easy processing as shown in Listing 2. A read query always starts from the base class, followed by filters, ordering, and aggregation, then retrieval of associated objects that are returned from subqueries. A subquery is similar to a read query except that it can only retrieve associated objects of the base class (e.g., issues that belongs to a project). For write queries, CHESTNUT rewrites them into queries that updates one object one at a time. As discussed in Section 1, OODAs often update a single object at a time, hence we leave the implementation of batch updates as future work.

```
readQ := Class.where(...).order(..).aggr(...)
                .includes(association, subQ)
writeQ := Class.(insert|update|delete)(id,...)
```

Listing 2: Stylized CHESTNUT queries with Rails APIs in bold.

4.2 Data layout search space

As mentioned in Section 3, CHESTNUT searches for the best data layout given a set of queries. A data layout describes how CHESTNUT-managed objects are stored in memory (e.g., as a simple or nested array), including indexes to be added. CHESTNUT customizes the search space for each set of queries. To make search effective, we represent the search space using the language shown below.

```
dataStructure   :=  topArray | nestedArray | index
topArray        :=  array(pred, key->value)
nestedArray     :=  nested_array(pred, key->value, nested)
index           :=  indexType(pred, key->topArray)
value           :=  obj | pointer(topArray) | aggr(obj)
indexType       :=  b-tree | hash
pred            :=  pred∧pred | pred∨pred | ¬pred
                    e==e |...| e in {e,...} | e between [e,e]|
                    exists(associatedObj, pred)
e               :=  constant | parameter | f
key             :=  {} | {f,...}
f               :=  field | (associatedObj.)+.field
```

In the above, `topArray`, `nestedArray`, and `index` are data structures that CHESTNUT enumerates during search. *Top-level arrays* store objects of a single class (e.g., `Project`), pointers to objects in another array,[5] or aggregation result. Top-level arrays can store only a subset of objects of the class (e.g., projects created earlier at a given time, as specified as the predicate `pred`), and the objects stored can also contain only a subset of its fields (e.g., those that are projected). Top-level arrays can also be sorted. The predicate `pred` can involve associated objects as defined by `has_many` association using `exists` (e.g., `Q1` in Listing 1), or associated objects defined by `has_one` as a normal class field (e.g., `Issue.where(project.name=?)`). The field used in the predicate or order key can be the field of object stored in the array or the field of an associated object, as defined by `has_one` association. If the array stores aggregated values, it may store a single aggregated value over a list of objects if `key` is `null` (e.g., `array(status='open',null→count(issues))` stores the count of all open issues), or aggregated values for each group (e.g., `array(status→count(issues))` stores an array of `(status, count)` pair for each status and its count).

Meanwhile, *nested arrays* store values of associated objects as an array within another object. They are the same as top-level arrays except that they store only objects associated with those in another nested or top-level array.

[4] A reconnaissance query is a read query that performs all the necessary reads to discover the set of objects to be updated given a write query, as introduced in [53].

[5] CHESTNUT currently does not support chained pointers.

Finally, *indexes* can be added to top-level arrays to map keys to object pointers. CHESTNUT supports B-tree and hash indexes created on all objects of one type, and also *partial indexes* on a subset of objects as determined by a predicate (e.g., all open issues).

As shown above, `topArray`, `nestedArray` and `index` are defined by three parameters: `pred` determines the set of objects to be included in the data structure, `key` is the sorted order of an array or index key, and `value` describes the objects or aggregation values stored. For instance:

```
topArray(exists(issues, status='open'),created→Project)
```

is a top-level array that stores `Project` objects containing open issues, and is sorted by the `created` field.

CHESTNUT's data layout is inspired by its query interface. First, because queries often filter objects of one class by examining their associated objects, CHESTNUT supports creating predicates and index/order keys using associated objects. Moreover, because queries often involve associated objects, CHESTNUT supports storing them as nested arrays. As we will see in Section 8, these features allow CHESTNUT to greatly improve the performance of OODAs as compared to traditional physical designers.

4.3 Query plan description

Given a data layout, CHESTNUT enumerates query plans to answer each query in the application. As CHESTNUT's data layouts include nested data layouts that are not supported by relational databases, CHESTNUT can no longer use their query plans for query execution. We instead design an intermediate representation to represent CHESTNUT's query plans as shown below. This intermediate representation is designed to describe high-level operations on the data structures such that the cost of a plan can be easily estimated, while facilitating easy C++ code generation.

$plan$	:=	$(s_{coll})^*$ $(s_{sdu})^*$ $(s_{setv})^*$ return v
$coll$	:=	ds.scan(k_1,k_2) | ds.lookup(k)
s_{coll}	:=	for v in coll: ($plan$)
s_{sdu}	:=	sort(v) | distinct(v) | v'=union(v_1,...)
s_{setv}	:=	if $expr(v)$ $setv$
$setv$	:=	v=$expr$ | v.append($expr$) | ds.insert(k,v) | ds.update(k,v) | ds.delete(k,v)
$expr$	:=	$v + v \mid v \wedge v \mid ... \mid *v \mid v.f \mid$const

Each read query plan starts with a `for` loop that looks up from an index or scans an array (s_{coll}, where ds is a data structure), followed by statements (s_{sdu}) that perform sort, distinct, or union, and returns a list of objects or aggregation result. The loop body conditionally appends objects into the result object list or set the value of variables like aggregation result (s_{setv}). Each s_{coll} loop may contain recursive subplans with nested loops that iterates over data structures storing associated objects.

Unlike relational databases that need to convert query results from relations to objects (i.e., deserialization), CHESTNUT's query plan returns objects and nested objects directly. Doing so reduces the overhead of time-consuming deserialization and allows CHESTNUT's query plan to be often faster than similar relational query plans that require deserilization.

Likewise, a write query plan also starts with an index lookup or array scan to find the object to update, followed by modification of each identified object. Updating a nested object can be slower than updating a relational tuple due to the overhead of locating the object. Besides, multiple copies of an object may be stored which makes a write query slower. However, since most updates in OODA are single-object updates, the overhead of slower writes are small compared to the gain from read queries, as we will see in Section 8.

5. QUERY PLAN GENERATION

We now discuss how CHESTNUT enumerates data layouts and query plans efficiently and finds the optimal one for each read query. After determining the best data layout, we discuss finding query plans for write queries in Section 6.

5.1 Enumerate data layouts

CHESTNUT first enumerates data layouts that can be used to answer a read query Q. This includes enumerating object representations, i.e., whether objects of one class are stored as top-level or nested objects, as well as data structures to be used. Each design is described using the language described in Section 4. Unlike traditional physical designers, CHESTNUT only enumerates data structures involving objects and predicates used in Q. As we will see, doing so greatly reduces the size of the search space.

Algorithm 1 Object nesting enumeration

```
1:  procedure ENUMERATEDS(Q,C,ds_upper=null)
2:      models←[] // data layouts to be returned
3:      DS_c ←[] // data structures to store objects of C
4:      for key∈ {scalar fields of C used in Q} do
5:          for pred∈{partial pred in Q with no param} do
6:              if ds_upper is null then
7:                  array← array(pred,key,C)
8:                  DS_c.add(array)
9:                  DS_c.add(B-tree(pred,key,ptr( array ))
10:                 DS_c.add(hash(pred,key,ptr( array ))
11:             else
12:                 DS_c.add(array(pred,key,C,ds_upper))
13:     for a∈{C's associated objects involved in Q} do
14:         Q_a ← sub-query/sub-predicate on a
15:         for an ∈ EnumerateDS(Q_a,ds) do
16:             for ds∈DS_c do
17:                 ds.addNested(an))
                    models.add(DS_c)
18:             for ds∈DS_c do
19:                 ds.addNested(an.replace(value,ptr))
20:             models.add(DS_c)
21:         for an ∈ EnumerateDS(Q_a) do
22:             models.add(an.addNested(DS_c))
23:     return models
```

The algorithm in shown in Algorithm 1. It takes Q as input, the class `C` to generate designs for, and optionally a design ds_{upper} where `C` is to be nested within (`C` is initially set to be the base class of Q and ds_{upper} is set to null). It first enumerates the data structures to store `C` and saves them in DS_c. On line 4, it enumerates `C`'s fields used in Q's predicates as the sort or index key for the data structure to be created, e.g., `created` in Q1, and on line 5 the predicates in Q that uses `C` and does not use input parameters, e.g., `exists(issues,status='open')` in `Q1`'s predicates from Listing 1. The data structures are then enumerated. If `C` is not to be nested in another data structure (line 6), it then creates a top-level array using the key and predicate (line 7), along with indexes on that array (line 8-10). Otherwise, `C` is nested in ds_{upper} (line 12).

After enumerating the data structures to store `C`, we enumerate the structures to store any associated class `A` of `C` that is used in Q. CHESTNUT considers three ways to store `A`, which we illustrate using the `Project-Issue` association in `Q1`:

1. Store `Issues` as a nested array within each `Project`. Issues of one project can then be retrieved by visiting these nested objects.

2. Same as 1. but store pointers to a separate array of `Issues`. In this case issues of a project are retrieved through pointers.

3. Store `Issues` as a top-level array, and keep a copy of project as a nested object within each `Issue`. Issues of one project p are then retrieved by visiting top-level `Issues` and finding those whose nested project matches p.

These three designs are generated from lines 15 to 22 in Algorithm 1. We iterate through the different data structures used to store C and combine that with data structures storing A. For every associated class A, the algorithm first extracts the subquery or the subquery's predicate from Q that uses A, e.g., the "status='open'" predicate on the associated Issues. It calls EnumerateDS using the sub-query or the predicate to recursively enumerate the data layouts for A and A's associated objects, and merges A into ds (i.e., the data structure that stores C) using one of the three ways mentioned above. The final layout is created by either nesting the associated objects (line 17), nesting pointers (line 19), or nesting the parent objects within the associated ones (line 22).

5.2 Enumerate query plans

Given the data layouts, CHESTNUT next enumerates query plans for them. These plans are represented using the intermediate language described in Section 4.3. Because of the huge variety of data layouts that can be generated for each OODA, CHESTNUT does not use rules to generate correct query plans as in traditional query optimizers. Instead, it enumerates as many plans as possible up to a bound, and verifies if any of them is correct.

To illustrate, suppose we have a query Q with no associations, written as `Class.where(pred).order(f,...)` as described in Section 4. Suppose we are given a storage model containing a list of data structures $\{ds_0, ds_1, \ldots\}$ storing objects of Q's base class C. CHESTNUT uses a skeleton that defines the "shape" of a query plan, and enumerates only plans whose number of statements is smaller than the number of disjunctions in Q's predicate.[6]

The skeleton is shown as below:

```
plan  :=  (op)+ (v'=union(v1,v2,...))?
          (distinct(v'))? (sort(v'))?
op    :=  for o in ds.scan/lookup(params):
          (if pred_r v.append(o))
```

The skeleton consists of several operations (op) followed by optional operations that merge and process the results from previous operations: `union`, `distinct`, or `sort`. Each op performs a scan or lookup on a data structure ds, and evaluates a predicate $pred_r$ over the input objects. The skeleton is instantiated by filling in the data structure ds, the predicate $pred_r$, and the number of operations op given the generated storage models.

If Q involves no associations, then all plans would be enumerated by the above. If not, CHESTNUT enumerates sub-plans pl_a to answer the predicate on the associated objects using the same process as above, and merges the sub-plans into the loop body of pl to produce the final nested-loop query plan for Q.

5.3 Verify query plans

We now have two ways to represent a query: as expressed using the Rails API as described in Section 4, or as one of the CHESTNUT-generated query plans expressed using the intermediate language described in Section 4.3. Our goal is to check the CHESTNUT-enumerated query plans to see which ones are semantically equivalent to the input, i.e., they return the same results.

Testing can be used to check for plan equivalence: we generate random objects, pass them as inputs to the two plans, execute the plans (using Rails for the input query, or as a C++ program for the CHESTNUT-generated plan), and check if the two plans return the same results. However, this is costly given the large number of plans enumerated by CHESTNUT. CHESTNUT instead leverages recent advances in symbolic reasoning [55, 36, 35] for verification.

[6] The idea is that each loop generated by op answers one one clause in the disjunction, and objects from different clauses are then unioned to generate the final result. If Q is a conjunctive query, then one loop would be sufficient to retrieve all objects.

This is done by feeding the two plans with a fixed number (currently 4) of *symbolic* objects but with unknown contents. Executing the plans with symbolic objects will return symbolic objects, but along with a number of *constraints*. For instance, given the following CHESTNUT-generated query plan that scans through an array of projects:

```
for p in projects.scan:
  if p.created < '1/1':
    v.append(p)
```

Executing this plan with an array of `projects` consisting of two symbolic objects `projects[0] = p0` with `p0 = {created=c0}` and `projects[1] = p1` with `p1 = {created=c1}` where the creation dates `c0` and `c1` are unknown will return the following constraints:

```
if c0 < '1/1' && c1 < '1/1':
  v[0] = p0  v[1] = p1
else if c0 >= '1/1' && c1 < '1/1':
  v[0] = p1  v[1] = null
else if c0 < '1/1' && c1 >= '1/1':
  v[0] = p0  v[1] = null
else # both projects are created >= '1/1':
  v[0] = null  v[1] = null
```

While we do not know the concrete contents of v after execution, we know the constraints that describe what its contents should be given the input. We execute the Rails and CHESTNUT-generated plan to generate such constraints, send them to a solver [23] to check for equivalence under all possible values of `c1` and `c2`, and retain only those plans that are provably equivalent. Checking such constraints can be done efficiently: in our evaluation it takes less than 1s to check when fewer than two associations involved.

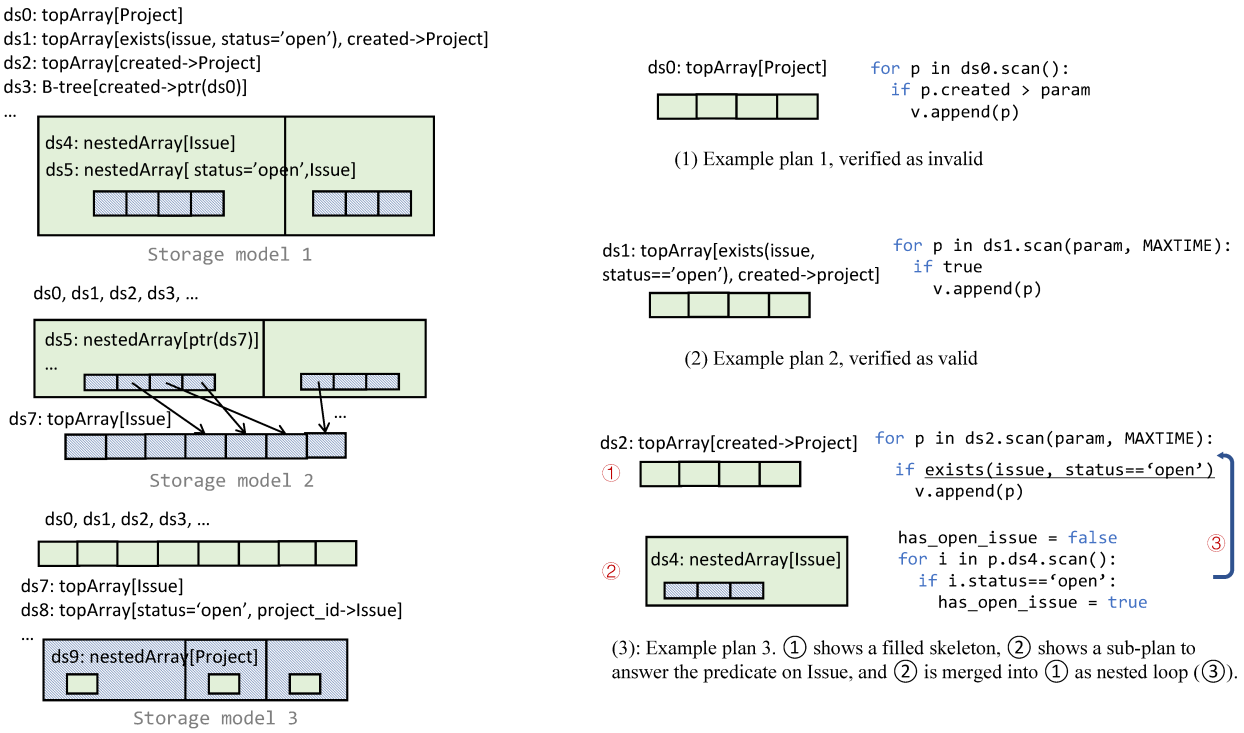
5.4 An illustrative example

We now use Q1 in Listing 1 to illustrate query plan enumeration, where it returns the `Projects` created later than a parameter `param` that contain open `Issues`.

CHESTNUT first enumerates the data structures to construct data layouts as described in Section 5.1. Figure 4(a) shows three data layouts. In the first one, `Issues` are nested within each `Project`. A few data structures to store `Projects` are shown from ds_1 to ds_3. The keys of a data structure are fields used in Q1 like `created` in ds_1, and the predicates are partial predicates from Q without parameters such as `exists(issue,status='open')` in ds_1. Different combinations of keys, predicates, and data structure types are enumerated. CHESTNUT similarly enumerates nested data structures to store `Issues` like ds_4 and ds_5.

Other data layouts store the association between `Projects` and `Issues` differently from layout 1. In layout 2, `Projects` store nested `Issue` pointers that point to a top-level `Issue` array ds_7, while in layout 3, top-level `Issues` store nested `Projects`. These three layouts shows the three ways to store associated class as described in Section 5.1. In layouts 2 and 3, CHESTNUT also enumerates data structures using different keys, predicate and data structure types, with some of them are shown in Figure 4(a).

After the data layouts are enumerated, CHESTNUT then generates query plans as described in Section 5.2. Figure 4(b) shows three examples of the enumerated plans. Since Q1's predicate has no disjunctions, CHESTNUT enumerates plans using the skeleton with up to one op, and fills op with different ds and $pred_r$. In (1), the skeleton is filled with ds_0 (the `Project` array that stores all `Projects`), and $pred_r$ as `p.created>param`. In (2), ds_1 and `true` are used. Both (1) and (2) are complete plans as $pred_r$s does not involve associated objects. In (3) however, the predicate `exists(issues, status='open')` involves associated issues (as shown in ①), so CHESTNUT generates a sub-plan that stores the nested `Issues` to



(a) Example storage models enumerated by Chestnut

(b) Example query plans enumerated given Q1 and storage model 1

Figure 4: Example of how CHESTNUT enumerates data structures and plans for Q1. A green box shows a `Project` and a blue box an `Issue`.

fill a skeleton, as shown in ②. The sub-plan is then merged into the loop of ① to form a complete plan.

Each plan is then verified against Q1 as described in Section 5.3. The solver determines that plan 1 is invalid and discarded (as it does not evaluate the partial predicate `exists(issues, status='open')` in Q1), while plan 2 and 3 are valid and retained.

5.5 Plan cost estimation

To find the optimal plan, CHESTNUT estimates the execution time of a plan using a cost model, and assumes that predicate selectivities are provided. The plan cost is then estimated as the number objects visited in the plan. For instance, the cost of the plan 1 shown in Figure 4(b) is $N_{projects}$ (i.e., number of `Projects`) as each `Project` is visited once when scanning ds_0; the cost of plan 3 is $N_{projects} * N_{issue\_per\_project}/2$ as the top-level loop visits $N_{projects}/2$ `Projects` (assuming the value of `created` is evenly distributed), and each nested loop visits $N_{issue\_per\_project}$ nested issues. Other cost models can be used with CHESTNUT as well.

5.6 Reducing search space size

CHESTNUT's enumeration algorithm produces a large number of query plans for each query. CHESTNUT uses three heuristics to prune away plans that are unlikely to be the optimal ones.

First, as mentioned in Section 5.1, CHESTNUT only enumerates layouts involving classes used in the queries. A layout storing an associated class `A` as top-level and `C` nested in `A` (e.g., layout 3 in Figure 4(a)) is discarded if the association is only used in one direction in the application, for instance, queries only retrieve `Issues` given `Projects` but not vice versa. Furthermore, if no query retrieves the associated objects directly, then any layout that stores them as top-level arrays (like layout 2 in Figure 4(a)) is also dropped.

Second, CHESTNUT drops plans that are subset of others. For example, if p performs the same scans on the same data structure ds twice while there exists another plan p' that only scans ds once, then p is redundant and is dropped.

Third, if the plans for one query have the same set of shared data structures, then CHESTNUT only keeps the best ones in terms of either least memory consumption or plan execution time. To do so, CHESTNUT groups the plans enumerated for one query by the set of data structures they share with other queries. For each group, CHESTNUT drops all plans other than those that has minimum execution time or uses a data layout with minimum memory cost.

These pruning heuristics can greatly reduce CHESTNUT's execution time, as we will discuss in Section 8.6.

6. HANDLING WRITE QUERIES

After enumerating plans for read queries, CHESTNUT generates plans for writes. As mentioned in Section 1, OODAs often make individual object updates, hence CHESTNUT currently supports single-object writes (i.e., insert, delete, or update), and translates multi-object writes into a sequence of single-object writes.

To generate plans for writes, CHESTNUT first executes "reconnaissance" read queries [53] to retrieve the objects that are updated. After that, CHESTNUT enumerates plans for these reconnaissance queries, whose results are then used to generate a write plan.

6.1 Generating reconnaissance queries

Given a write query Q_w on object o of class `C` and a data structure ds, ds will be updated if ds's predicate, key, or value involves `C`, or ds contains nested object of class `C`. For example, if an `Issue` is updated, `Issue` arrays and indexes, data structures whose key uses `Issue`'s field and arrays containing nested `Issue` will be updated.

CHESTNUT generates up to two reconnaissance queries for each ds. The first retrieves the affected object in ds using o's id (denoted as o_{ds}), and the second retrieves associated objects of o_{ds} to compute the new key and predicate of o_{ds}. We use Q2 updating `ds1` as shown in Section 3 to illustrate. The first query (`Qw1` below) retrieves the project in `ds1` containing the issue to be updated. The second one (`Qw2`) retrieves that project's issues to recompute if that project contains any other open issues.

```
Qw1=Project.where(exists(issues, id=?))
Qw2=Project.where(id=?).includes(issues)
```

Listing 3: Read query to find the project of a deleted issue

6.2 Generate plans for write query

CHESTNUT then uses the same search procedure described in Section 5 to find plans for the read queries generated, and constructs a write plan using the read results.

The algorithm to generate write plans is shown in Algorithm 2. Lines 5-6 generate the two read queries as described above. It then finds plans for these queries using the same process shown in Section 5 (line 7-8). These read queries find all the entry objects in ds that need to be updated. For each entry object, it first deletes the entry from ds (line 12), updates the entry and recomputes the key, predicate and value (line 13), and reinserts it into ds (line 14).

Algorithm 2 Generate plan for Q_w to update ds

```
1:  procedure GENWRITEPLAN(Q_w, ds)
2:      plan←[]
3:      if not isAffected(ds) then
4:          return
5:      Qw1← ds.class.where(..., Q_w.id)
6:      Qw2← ds.class.where(..., obj.id)
7:      plan.add_stmt((EnumeratePlan(Qw1))
8:      plan.add_stmt((EnumeratePlan(Qw2))
9:      objs←plan.result
10:     for o ∈ objs do
11:         o.update()
12:         ds.delete(o)
13:         (k, pred, v)←(ds.key(o),ds.pred(o),ds.value(o))
14:         plan.add_stmt(if pred ds.insert(k,v))
15:     return plan
```

CHESTNUT first generates an update plan for every ds enumerated for all input read queries. In this process it generates new reconnaissance queries, whose data layouts may contain new data structures not used by any input read query, thus do not have corresponding update plans yet. CHESTNUT iteratively produces write plans to update these new data structures until no new data structures and all write plans are added. In practice, this process converges quickly, usually within just a few rounds.

7. FINDING SHARED DATA STRUCTURES

The above describes CHESTNUT's data layout and plan enumeration for all queries in the OODA. By formulating into an integer linear programming (ILP) problem, we now discuss how CHESTNUT selects the best data layout from them by trading off between query performance and memory usage.

The ILP consists of the following binary variables:

- each data structure is assigned a variable ds_i to indicate whether it is used in the chosen data layout.
- if ds_i stores an object of class C, then each of C's field f is assigned a variable $f_{ds_i[f]}$ to indicate if f is stored in ds_i (recall that a data structure may store only a subset of an object's fields).
- each plan for read query i is assigned a variable p_{ij} to indicate if plan j is used for query i.
- each plan for write query i updating a data structure ds is assigned a variable p^w_{ijd} to indicate if plan j is used to update the data structure ds.

CHESTNUT estimates the memory cost of each data structure ds_i, denoted as C^{ds}_i, as well as the execution time of each query plan C^{ds}_{ij} or C^{p}_{ijd} (for read or write plans, respectively) as described in Section 5.5. It also calculates the size of field f as $F^{ds[f]}$, and estimates the number of elements in ds_i as N^{ds}_i.

CHESTNUT then adds the following constraints to the ILP:

- Each read query uses only one plan: $\sum_j p_{ij} = 1$
- Each write query uses only one plan: $\sum_j p^w_{ijd} = 1$.
- Plan p_{ij} uses all the data structures $ds_1, \ldots, ds_N$ in its plan.

Similar constraints are created for write query plans: $p_{ij} \rightarrow ds_1 \wedge \ldots \wedge ds_N$.

- Plan p_{ij} uses object fields $f_{ds[t_1]}, \ldots, f_{ds[t_N]}$ in array ds in the plan. A field is used when the plan has a s_{setv} statement that evaluates a predicate, computes an aggregation that involves that field, or adds an object to the result set where the field is projected. Similar constraints are created for write query plans: $p_{ij} \rightarrow f_{ds[t_1]} \wedge \ldots \wedge f_{ds[t_N]}$.
- If a data structure ds_k is affected by a write query Q_j, then at least one update plan should be used: $ds_k \rightarrow \wedge_j(\vee_i P^w_{ijk})$.
- The total memory cost of used data structures and object fields should be smaller than the user-provided bound M: $\sum_i D_i * C^D_i + \sum_{ij} N^D_i * F^{f_j}_{D_i} \leq M$.

The objective is then to minimize the total execution time of all queries:

$$(\sum_i \sum_j C^P_{ij} P_{ij} w_i) + (\sum_i \sum_d \sum_j D_d C^P_{ijd} P^w_{ijd} w_i)$$

where the sums are the total execution time of all read and write queries, respectively. Each query i is associated with a weight w_i to reflect its execution frequency in the OODA, which can be provided by developers or collected by a profiler.

We use the example shown in Figure 4 to illustrate. Assume the application contains `Q1` and `Q2` shown in Listing 1. Due to space, we only show two plans for `Q1` (shown in Figure 4(b)) and a subset of data structures enumerated by CHESTNUT (ds1, ds2 and ds3). The following lists a subset of the ILP constraints, while the rest involving other plans and data structures are constructed similarly.

- Q1 uses only one plan: $p_{11} + p_{12} + ... = 1$.
- Q2 uses only one plan to update ds1: $p^w_{211} + p^w_{212} + .. = 1$.
- Q1's plan1 uses ds1: $p_{11} \rightarrow ds_1$.
- Q1's plan2 uses ds2 and ds3: $p_{12} \rightarrow ds_2 \wedge ds_3$.
- Q1's plan2 uses fields `created` in ds2 and `status` in ds3: $p_{12} \rightarrow f_{ds_2[created]} \wedge f_{ds_3[status]}$.
- ds1 is updated by Q2: $ds_1 \rightarrow p^w_{211} \vee ....$
- ds3 is updated by Q2: $ds_3 \rightarrow P^w_{231} \vee ....$
- Total memory consumed does not exceed bound: $ds_1 * C^{ds}_1 + ds_2 * C^{ds}_2 + ... + f_{ds_2[created]} * N^{ds}_2 + f_{ds_3[status]} * N^{ds}_3 ... \leq M$.
- **Goal**: $\min(p_{11} * C^p_{11} * w_1 + p_{12} * C^p_{12} * w_1 + ...)$

The ILP's solution sets a subset of variables of ds_i, $f_{ds_i[f]}$, p_{ij}, p^w_{ijd} to 1 to indicate the data structures and plans to use, along with a subset of fields to store in each data structure. CHESTNUT then generates C++ code for the chosen plans and data structures using the STL library and Stx-btree library [19]. For query plans, it translates each IR statement in the query plan into C++. We omit the details due to space.

8. EVALUATION

CHESTNUT is implemented in Python with gurobi [9] as the ILP solver, and uses the static analyzer described in [57] to collect the object queries. We evaluate CHESTNUT using open-source Rails OODAs.

8.1 Experiment setup

8.1.1 Application corpus

Similar to prior work [58], we categorize the top-100 starred Rails applications on GitHub into 6 categories based on their tags. We then pick 4 categories, project management, chatting service, forum, and web scraping, and pick one of the top-2 applications

from each category as our evaluation corpus. All of the four chosen applications are professionally developed. They are selected as "trending" Rails applications [16, 20, 4], and widely used by companies, individual users [22, 21] and prior research [39, 59].

• *Kandan* [12] is an online chatting application structured with classes `User`, `Channel`, `Activity`, etc. `Users` can send messages in `Channels`. Each `Channel` lists `Activitys`, where an activity is often a message. Queries in this application contain simple predicates, but retrieve a deep hierarchy of nested objects. For example, Kandan's homepage lists channels with their activities and creators of each activity. The corresponding data is retrieved using a query that returns a list of `Channels` with nested `Activitys`, where each `Activity` contains nested `Users`.

• *Redmine* [17] is a collaboration platform like GitHub structured around `Project`, `Issue`, etc. Each project belongs to a `Module` and contains properties like issue tracking. The `Tracker` class is used to manage what can be tracked in a project, such as issue updates, and has a many-to-many relationship with `Projects`.
This relationship is maintained by a mapping table with two columns, `project_id` and `tracker_id`. Such associations result in many-way materialized joins as well as complex predicates predicates to retrieve the associated objects. In addition, Redmine's queries also use disjunctions extensively. For instance, it uses nested sets [14] as trees to organize each project and its children projects. A query that retrieves projects in a subtree with ID within the range (`p1`, `p2`) is done using the predicate `left>=p1 OR right<=p2`, where `left`, `right` `Project`'s fields. Such range predicates are often combined with others and are difficult for relational databases to optimize well. We will discuss this later in the evaluation.

• *Lobsters* [13] is a forum application like Hacker News. Persistently stored classes include `User`, `Story`, `Tag`, etc. Users share URLs as stories and add tags to them. Lobsters has similar query patterns as *Redmine*, with many-to-many associations. For example, a `Story` has many `Tags`, and the same `Tag` can be added to many `Storys`. As a result, many queries are multi-way joins.

• *Huginn* [11] is a system to build agents to perform automated web scraping tasks. It persistently stores `Agents`, each with a set of `Delayed_jobs` to automatically run in the backend to watch these `Agents`, and records `Events` when any update happens. Its queries retrieve a hierarchy of nested objects, as well as aggregations to render the current state of an `Agent` in various ways.

Table 1: Application statistics

Application	# of classes	# read queries	# write queries
Kandan	6	10	6
Redmine	12	24	6
Lobsters	7	26	10
Huginn	6	16	7

We select the top-10 most popular pages from each application. These pages are chosen by running a crawler from prior work [58]. The crawler starts from the application's homepage and randomly clicks on links or fills forms on a current page to go to a next page. This random-click/fill process is repeated for 5 minutes, and we collect the top 10 mostly-visited pages. Table 1 shows the total number of classes and distinct read and write queries involved in the popular pages. We collect the queries executed when generating these pages from the SQL query log, and trace back to the object queries defined in the application. These object queries then serve as input workload to CHESTNUT.

We populate the application's database with synthetic data using the methodology described in [58]. Specifically, we collect real-world statistics of each application based from its public website or similar website to scale data properly as well as to synthesize database contents and application-specific constraints. We scale the size of the application data to be 5GB-10GB, which is close to the size of data reported by the the application deployers [5, 6].

8.1.2 Baseline database engines

We compare the CHESTNUT-generated database with relational in-memory databases, including MySQL, PostgreSQL and a commercial database.

• **MySQL** (version 5.7). The original applications use MySQL as the default backend database, so we use the same setting as the baseline and add the same indexes that the developers specified in the application.

• **PostgreSQL** (version 12). We use an indexing tool [7] to analyze the query log and automatically add more indexes besides the ones that come with the application. We set the buffer pool size for MySQL and PostgreSQL to be larger than the total of data and indexes (20GB) such that data stays in memory as much as possible. We use the default value for other settings.

• **System X**. We further use a commercial high-performance in-memory column-store database as the third comparison target. We add the same set of indexes we use in PostgresSQL as suggested by the automatic indexing tool.

• CHESTNUT. When using CHESTNUT, we measure the actual total size of tables and indexes used by PostgreSQL and set it as the memory bound for CHESTNUT. We replace the database engine and Rails' serialization code with CHESTNUT, and connect to a backend MySQL database to persist data.

We measure the time from issuing the query until the result converted into Ruby objects as the query time. Hence query time includes both **data-retrieving time**, i.e., the time to execute the queries, and **deserialization time**, i.e., the time to convert query results into ruby objects. For relational databases, Rails deserializes the query result from relational tables into (nested) Ruby objects. CHESTNUT's query plans return results as C++ (nested) objects, and uses protocol buffer [15] to convert them into Ruby objects.

All evaluations are performed on a server with 128 2.8GHz processors and 1056GB memory. In our current prototype CHESTNUT does not support transactions, so we run queries sequentially and measure their latency.

8.2 Performance Comparison

Figure 5 shows the performance comparison across OODAs. We focus on slow queries takes longer than 100ms to execute in the original application setting as they are often performance bottlenecks. To get a better understanding of such queries, we show the breakdown of data retrieval and deserialization time in Figure 5. We show only the summary, i.e., the min, max and mean times of the remaining read and write queries as they execute quickly.

The results show that using the automatic indexing tool improves the performance for half of the slow queries by up to 163×. Interestingly, unlike OLAP queries where in-memory column stores can substantially accelerate queries, System X does not achieve similar speedup for OODA's queries. As discussed in Section 1, this is because query results in OODAs are often not aggregated but returned as lists of objects where all columns are projected. Rather than speeding up queries, columnar store instead adds the non-trivial overhead of row materialization. Compared to other relational database engines, CHESTNUT shows better performance in all slow queries, with an average speedup of 6×, 3.9×, and 3.6× against the MySQL, PostgreSQL, and System X respectively.

The memory consumption comparison is shown in Table 2. It shows the size of the application data and the maximum runtime memory used by PostgreSQL (e.g., tables, indexes and intermediate tables) and CHESTNUT's engine (e.g., data layout). Despite

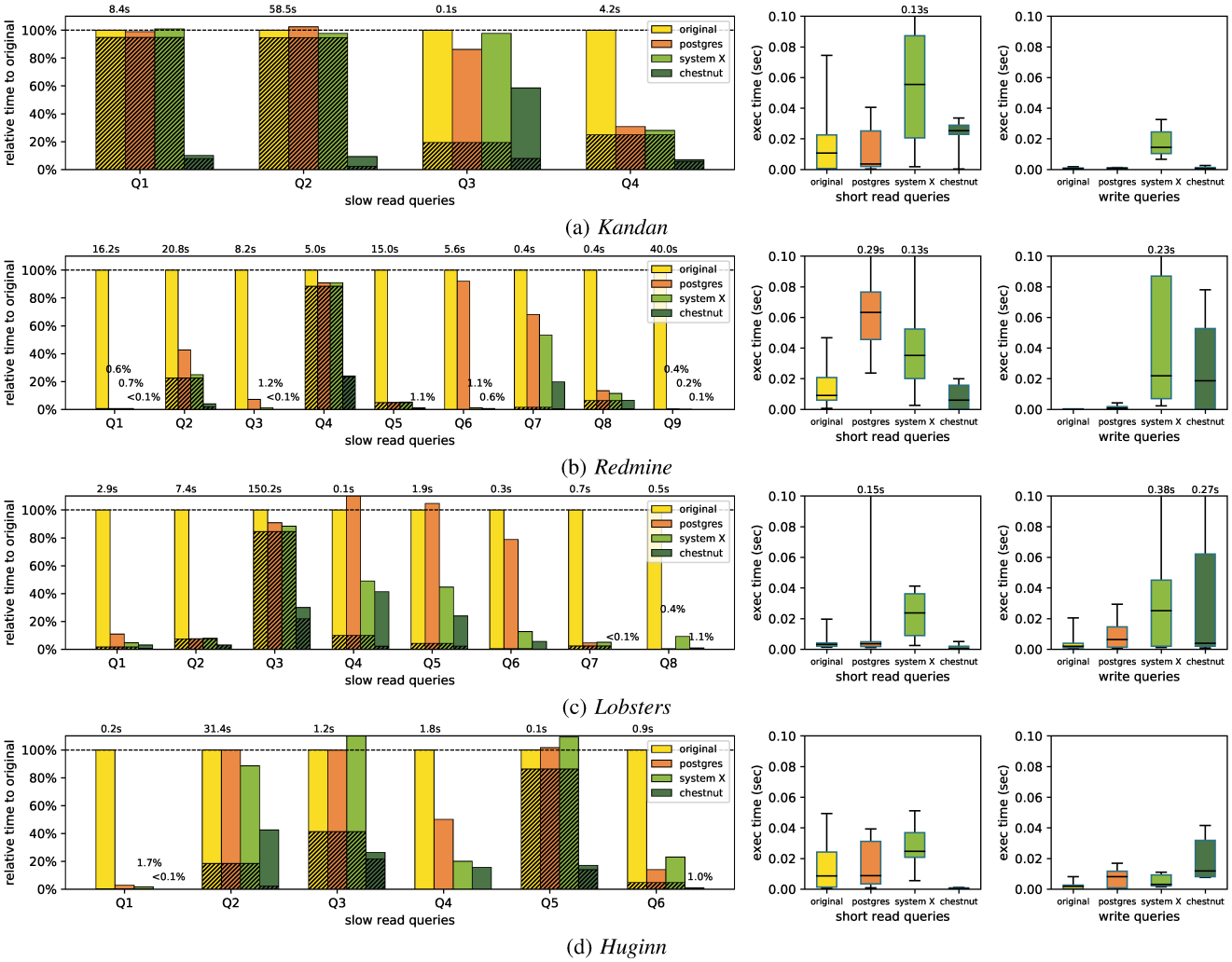


Figure 5: Query performance comparison. Figures on the left shows the relative time of slow queries compared to the original. The original query time is shown as text on the top. Each bar is divided into the top part showing the data-retrieving time and the shaded bottom part showing the deserialization time. Figures on the right shows the summary of other short read queries and write queries.

Table 2: Comparison of runtime memory cost

App	data size	PostgreSQL	CHESTNUT
Kandan	5.1GB	7.2GB	5.8GB
Redmine	5.9GB	8.3GB	7.4GB
Lobsters	9.8GB	14.5GB	13.7GB
Huginn	6.0G	9.8G	9.0G

using smaller amount of memory than PostgreSQL, CHESTNUT still achieves significant performance gain in comparison, as shown in Figure 5.

We next present case studies to analyze why CHESTNUT's data layout delivers better performance on OODA's queries.

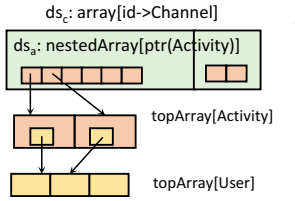
Case 1: Kandan-Q1. This query, shown in Figure 6(a), retrieves the first 50 `Channels` ordered by its `id`, where each `Channel` includes its associated `Activitys` and each `Activity` includes its associated `User`. The Rails-generated SQL queries are shown in Figure 6(b). They first select from the `channel` table, then use the foreign key `channel_id` to select from the `activity` table, followed by a third selection from the `user` table using the values of the `user_id` column in the second query result as parameter. The cost of combining the tuples from these tables to generate the final query results is prohibitively expensive. Rails would 1) create `Channel` objects from the result of the first query, 2) create `Activity` objects from the second query, 3) group `Activity` by `channel_id` and insert into `Channel` as nested object, 4) similarly create nested `User` object inside each `Activity`. In our evaluation, with one channel having 10K activities on average, deserialization alone takes 55s to finish, which is much longer than query execution time.

CHESTNUT instead generates a layout that stores `Channel` objects in an array sorted by `id`, a nested array of associated `Activity` pointers within each `Channel` object, and `User` pointer in each `Activity`, as shown in Figure 6(c). To answer this query, CHESTNUT's query plan scans the `Channel` array. Inside this scan loop, it uses a nested loop to retrieve `Activity` objects via pointers stored in each `Channel`, and then retrieves `User` objects via pointer stored in each `Activity`. Although such nested-loop query plan is slower than System X (7.5s compared to 1.7s), skipping object materialization greatly reduces deserialization time from 55s to 1.5s.

Case 2: Redmine-Q3. This query, shown in Figure 9(a), retrieves the `Trackers` that relate to active projects that contains issue-tracking module. Rails generates a query that contains three joins as shown in (b). The first two joins retrieve the associated `Projects` for each `Tracker` through the mapping table `project_tracker`. The third join is a semijoin that selects `Projects` within the issue-tracking module. Since the inner joins produce a denormalized table with

```
Channel.includes(activities, includes(user))        (a)
       .order(id).limit(50)
```

```
SELECT * FROM channel ORDER BY id LIMIT 50;
SELECT * FROM activity WHERE channel_id IN (…);      (b)
SELECT * FROM user WHERE id IN (…);
```




```
for c in ds_c.scan:
  c_0 = c
  for a in c.ds_a.scan:                              (c)
    a_0 = *a
    a_0.user = *((*a).user)
    c_0.activities.append(a_0)
  v.append(c_0)
return v
```

Figure 6: Case study of Kandan-Q1. (a) shows the original query. (b) shows the corresponding SQL queries. (c) shows the data layout (left) and the query plan (right) generated by CHESTNUT.

duplicated `trackers`, the query then groups by `Tracker`'s id, followed by `DISTINCT` to generate a list of unique trackers. This many-way join takes over 8s to finish in the original application, 592ms after indexes are added in PostgreSQL, and 98ms with System X.

As this query does not have user-provided parameters, its results can be pre-computed. The CHESTNUT-generated data layout does exactly this: it stores the `Trackers` that satisfy the query predicate into an array such that the query plan only scans this array, as shown in Figure 9(c). Doing so reduces the query time to only 0.2ms.

However, the data layout chosen by CHESTNUT brings extra overhead to update queries. A query that removes an issue-tracking module from a project slows down from 1ms to 76ms, since the write query plan now needs to re-examine a tracker's projects to see if that tracker contains other active, issue-tracking-enabled projects other than the one to be removed. However, the write query's overhead is relatively small comparing to the read queries, and the overall application performance application is still improved.

Case 3: Redmine-Q8. The query is shown in Figure 10(a). It retrieves projects of a user-provided status in a subtree whose range is defined by the `left` and `right` parameters. When the selectivities of predicate "`status=? AND left>=?`" and "`status=? AND right<=?`" are small, indexing on `(status, left)` and `(status, right)` can accelerate the query compared to a full table scan. CHESTNUT-generated data layout creates these two indexes on `Projects`. The corresponding query plan performs a range scan on each index and unions the results, taking only 69ms to finish. Query optimizers in MySQL, PostgreSQL, and System X are unable to use such indexes for this query. Their query plans scan the entire `project` table even when the indexes are created, resulting in over 180ms to finish for all engines. Our investigation reveals that the plans will leverage these indexes only when the query is rewritten using `UNION` instead of `OR`, which is not how Rails generates queries. In contrast, CHESTNUT uses custom enumeration and symbolic reasoning to find query plans rather than looking for particular query patterns to optimize. As a result, it finds better query plans as compared to the relational engines.

8.3 Comparison with materialized views

We next compare CHESTNUT's data layouts with materialized views (MVs). We use MVs to optimize the slow queries executed using relational database engines even after indexes are added. For each of these queries, we manually create different combinations of views and indexes in System X to the best of our abilities and pick those that give the best performance. Since MVs are designed to optimize a single query, it is often a union of all query results under different parameter values, indexed by the fields that are involved in the query to compare to user-provided parameters. For example, the best MV for Q1 shown in Listing 1 is a table of all `Projects` containing open issues, with a clustered index created on field `created`. We measure the amount of memory used to create MVs for each query, and use it as the memory bound for CHESTNUT.

Both MVs and CHESTNUT selectively determine which subset of data to store, but MVs still use tabular layout and return relational query results. Instead, CHESTNUT chooses from both tabular and nested data layouts, as well as their combinations, and generate query plans that return objects. When using MVs, the bottleneck for slow queries is again object deserialization. Although MVs greatly accelerate the relational query, the overall query time is still dominated by deserialization. CHESTNUT's query plan instead returns nested objects directly without changing the data representation, significantly reducing deserialization time. The result is shown in Figure 7, where CHESTNUT's query plan outperforms its relational counterpart by 3.69× on average.

8.4 Scaling to larger data sets

In this experiment we show how CHESTNUT performs when we scale the application's data. We scale *Kandan*'s data to 50GB, and the evaluation result is shown in Figure 11. With larger data sets, the time spent in deserialization becomes more dominant, and CHESTNUT's ability to speed up data deserialization becomes very significant. Over the four slow read queries, CHESTNUT achieves an average speedup of 9.2× compared to MySQL, 6.6× to PostgreSQL and 6.5× to System X on the 5GB dataset, but speedup increases to 20×, 12.3×, and 12× respectively once data increases to 50GB.

8.5 Evaluation on TPC-H

To isolate the effect of data deserialization, we next evaluate CHESTNUT using eight analytical queries from the TPC-H [54] benchmark. Most TPC-H queries can be expressed using the Rails API in the stylized form as shown in Listing 2. We use CHESTNUT to find the best data layout for each query and compare the performance with System X (column store without indexes). Since these queries do not return objects, we do not include deserialization when measuring the query time, thus allowing us to study the quality of CHESTNUT-generated data layouts and plans.

The result is shown in Figure 8. CHESTNUT-generated database is slower in Q3 and Q6. In comparison to System X's column-oriented data store and custom machine code generation, CHESTNUT uses a slow sort from C++ STL and table scans over C++ vectors. For other queries, however, CHESTNUT is more efficient due to the better layout it finds compared to columnar tables. For example, a partial index on Q5 reduces the query time by over 90% because Q5's predicate involves many joined relations where the join predicates can be pre-computed because they do not involve user-provided parameters.

8.6 Search and verification time

We run CHESTNUT on a machine with 256 cores and 1056GB memory using 32 threads, and measure the time used to find the best data layout.

Table 3 shows the total number of query plans enumerated by CHESTNUT (before and after pruning) and the time taken to find the best layout. The total running time includes plan enumeration and verification (described in Section 6), as well as ILP solving (described in Section 7). CHESTNUT enumerates a large number of plans even for a few queries, up to millions. With the pruning optimization described in Section 5.6, however, the number is greatly reduced by 53-96%. This reduction makes the ILP solving finishes quickly, as quick as 3min in average.

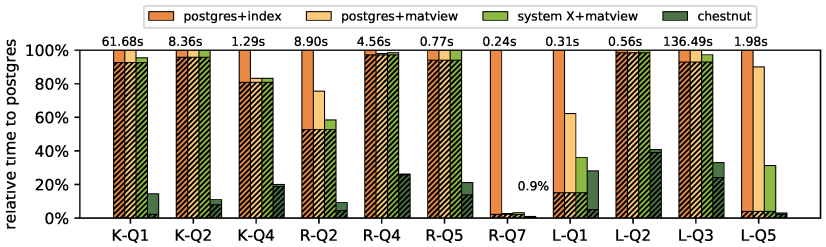


Figure 7: Comparison with hand-tuned materialized views on individual slow queries.

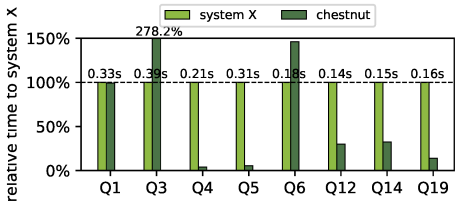


Figure 8: Evaluation on TPC-H.

```
Tracker.where(exists(projects, status!='inactive'
    AND exists(modules, name='issue_tracking')))
```
(a)

```
SELECT DISTINCT * FROM tracker t INNER JOIN project_tracker pt
  ON pt.tracker_id = t.id
  INNER JOIN project p ON p.id = pt.project_id WHERE
  (p.status != 'inactive' AND EXISTS
    (SELECT 1 AS one FROM module m
    WHERE m.project_id = p.id AND m.name='issue_tracking'))
GROUP BY t.id;
```
(b)

ds_t: array[exists(projects, status!='inactive' AND exists(modules, name='issue_tracking')), Tracker]

```
for t in ds_t.scan:
  v.append(t)
return v
```
(c)

Figure 9: Case study of Redmine-Q3.

```
Project.where(status=? AND (left>=? OR right<=?))
```
(a)

```
SELECT * FROM project WHERE status=? AND
  (left>=? OR right<=?)
```
(b)

ds_{p1}: btree[(status,left)->Project]
ds_{p2}: btree[(status,right)->Project]

```
for p in ds_p1.scan:
  v_0.append(p)
init(v_1)
for p in ds_p2.scan:
  v_1.append(p)
v=distinct(union(v_0,v_1))
return v
```
(c)

Figure 10: Case study of Redmine-Q8.

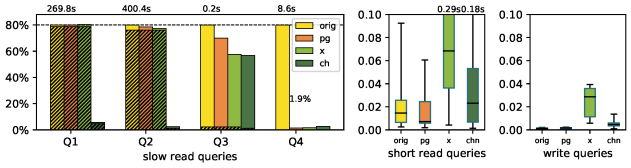


Figure 11: Scaling *Kandan*'s data to 50G.

While solving ILP is fast, the majority of CHESTNUT's running time lies in plan enumeration and verification. This part varies greatly depending on the query pattern. It becomes slow when a query involves many associations. An association between classes C_1 and C_2 involved in a query predicate produces a large symbolic expression to encode the mapping. When many associations involved, the number of mapping expressions grows exponentially. Verifying complicated expression takes longer and slows down plan synthesis. For a query involving six associations, verifying one plan can take over 5min. Fortunately, verification can often be done in parallel, as discussed in Section 5.3.

Plan synthesis can also be slow when the query predicate involves many disjunctions (e.g., many ORs are involved). As plan synthesis enumerates all plans from small a size up to a bound determined by the number of disjunctions (described in Section 5.2), lots of disjunctions leads to a large upper bound, hence many more plans to be enumerated and verified.

Meanwhile, the reason for CHESTNUT's relatively longer running time on TPC-H and *Lobsters* is due to the two reasons mentioned above. Some queries in TPC-H involve many associations, for instance, Q5 involves six associations; some has multiple disjunctions. *Kandan* has the shortest running time because it is a small application with the fewest number of classes and associations, and the query predicates are simpler. For *Redmine*, although it also has queries with disjunctions and associations, it is fast because many sub-queries in *Redmine* shares the same predicate. CHESTNUT caches the plans synthesized for each query and sub-query, and reuses the plans when seeing the same (sub-)query without synthesizing from start, so the running time is shorter than TPC-H and *Lobsters* which barely have any shared sub-queries.

Table 3: The number of query plans before and after pruning, and CHESTNUT's running time.

App	# query plans		running time			unoptimized mode
	Orig	prune1+ prune2	plan enum	ILP solve	total	# query plans
Kandan	405	191	<1min	<1min	1min	34
Redmine	78K	5K	9min	1min	10min	46
Lobsters	2031K	166K	42min	12min	54min	117
Huginn	9K	880	3min	<1min	3min	41
TPC-H	143K	667	48min	<1min	48min	43

Despite pruning, CHESTNUT still takes a while to compile a few applications. To help developers view and test data layouts quickly, CHESTNUT can run in an "unoptimized" mode where the search space of data layouts is restricted to use only row-major tabular layout with foreign key indexes. For all applications, CHESTNUT enumerates much fewer plans (as shown on the last column of Table 3) and the total running time under this mode is less than 1 minute. We envision developers running CHESTNUT in unoptimized mode to get a preliminary design, and rerun CHESTNUT in full mode once their application is ready for deployment to obtain the best results.

8.7 Discussion and future work

CHESTNUT currently has a few limitations. First, it does not handle concurrency control and multi-threading. Second, CHESTNUT's data layout may not be able to adapt to code and data distribution changes, and will need to recompile the application from the start. While incremental data layout update is an interesting work, we believe running CHESTNUT under the "unoptimized" mode will help developers get data layouts quickly during development time.

9. CONCLUSION

In this paper we presented CHESTNUT, a generator for in-memory database for OODAs. CHESTNUT searches for customized different types of data layouts to improve the performance of queries. Our experiments show that CHESTNUT can improve application's query performance by up to 42× using real-world applications.

10. ACKNOWLEDGEMENTS

This work is supported in part by the NSF through grants IIS-1546083, IIS-1651489; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; and gifts from Adobe, Google, and Intel.

11. REFERENCES

[1] Active record association. https://guides.rubyonrails.org/association_basics.html.

[2] Active record query interface. https://guides.rubyonrails.org/active_record_querying.html.

[3] Comparing oodb with LINQ. https://msdn.microsoft.com/en-us/library/aa479863.aspx.

[4] Curated list of awesome rails lists. https://project-awesome.org/ekremkaraca/awesome-rails.

[5] Database size reported by web developers. https://meta.discourse.org/t/slow-sql-queries/16604.

[6] Database size reported by web developers. http://www.redmine.org/issues/23318.

[7] Dexter, an automatic indexer for postgres. https://github.com/ankane/dexter.

[8] Django, a python web application framework. https://www.djangoproject.com/.

[9] Gurobi ilp solver. http://www.gurobi.com/.

[10] Hibernate, an orm framework for java. http://hibernate.org/orm/.

[11] Huginn, an agent monitor. https://github.com/huginn/huginn.

[12] Kandan, a chatting application. https://github.com/kandanapp/kandan.

[13] Lobsters, a forum application. https://github.com/lobsters/lobsters.

[14] Nested set model, a model to represent tree using relational table. https://en.wikipedia.org/wiki/Nested_set_model.

[15] Protocol buffer, a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers.

[16] ranking of popular open source rails applications. http://www.opensourcerails.com/.

[17] Redmine, a project management application. https://github.com/redmine/redmine.

[18] Ruby on rails, a ruby web application framework. https://rubyonrails.org/.

[19] Stx-btree library. https://github.com/bingmann/stx-btree.

[20] top7 forceful rails apps for business management. https://www.cleveroad.com/blog/effective-ruby-on-rails-open-source-apps-to-help-your-business.

[21] who uses huginn. https://github.com/huginn/huginn/wiki/Companies-and-People-Using-Huginn.

[22] who uses redmine. http://www.redmine.org/projects/redmine/wiki/weareusingredmine.

[23] Z3 theorem prover. https://github.com/Z3Prover/z3.

[24] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *PVLDB*, pages 496–505, 2000.

[25] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[26] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, pages 1103–1114, 2014.

[27] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. *The object-oriented database system manifesto*. Elsevier, 1990.

[28] T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *PVLDB*, 11(3):324–337, 2017.

[29] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the open oodb query optimizer. In *SIGMOD*, pages 287–296, 1993.

[30] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, 2005.

[31] N. Bruno and S. Chaudhuri. Constrained physical design tuning. *PVLDB*, 19(1):4–15, 2008.

[32] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *PVLDB*, pages 146–155, 1997.

[33] A. Cheung and O. Arden. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[34] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14, 2013.

[35] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[36] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524, 2017.

[37] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6):362–372, 2011.

[38] L. Fegaras and D. Maier. An algebraic framework for physical OODB design. In *DBLP*, pages 6–8, 1995.

[39] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *USENIX*, pages 213–231, 2018.

[40] D. Gluche and M. Scholl. Physical design in OODBMS. In *Grundlagen von Datenbanken*, pages 21–25, 1996.

[41] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[42] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

[43] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[44] W. Kim and F. H. Lochovsky. *Object-oriented concepts, databases, and applications*. ACM Press/Addison-Wesley Publishing Co., 1989.

[45] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. Coradd: Correlation aware database designer for materialized views and indexes. *PVLDB*, 3(1):1103–1113, 2010.

[46] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[47] Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[48] C. Loncaric, M. D. Ernst, and E. Torlak. Generalized data structure synthesis. In *ICSE*, 2018.

[49] C. Loncaric, E. Torlak, and M. D. Ernst. Fast synthesis of fast collections. In *PLDI*, pages 355–368, 2016.

[50] R. Marcus and O. Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *CIDR*, 2019.

[51] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. Nose: Schema design for nosql applications. In *ICDE*, pages 181–192, 2016.

[52] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *DEEM*, pages 4:1–4:4, 2018.

[53] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[54] Transaction Processing Performance Council. The TPC-H benchmark. http://www.tpc.org/information/benchmarks.asp, 1999.

[55] C. Wang, A. Cheung, and R. Bodík. Speeding up symbolic reasoning for relational queries. *PACMPL*, 2(OOPSLA):157:1–157:25, 2018.

[56] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. *PVLDB*, 9(5):444–455, 2016.

[57] C. Yan, J. Yang, A. Cheung, and S. Lu. Understanding database performance inefficiencies in real-world web applications. In *CIKM*, 2017.

[58] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung. How not to structure your database-backed web applications: A study of performance bugs in the wild. In *ICSE*, pages 800–810, 2018.

[59] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung. PowerStation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *ESEC/FSE*, pages 884–887, 2018.