Demonstration of Chestnut: An In-memory Data Layout Designer for Database Applications

Mingwei Samuel
Alvin Cheung
{mingwei,akcheung}@berkeley.edu
UC Berkeley

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

ABSTRACT

This demonstration showcases Chestnut, a data layout generator for in-memory object-oriented database applications. Given an application and a memory budget, Chestnut generates a *customized* in-memory data layout and the corresponding query plans that are specialized for the queries issued by the application.

Our demo will let users design and improve simple web applications using Chestnut. Users can view the Chestnut-generated data layouts using a custom visualization system, which will allow users to see how the application parameters affect Chestnut's design. Finally, users will be able to run queries generated by the application via the customized query plans generated by Chestnut or traditional relational query engines, and can compare the results and observe the speedup achieved by the Chestnut-generated query plans.

ACM Reference Format:
Mingwei Samuel, Alvin Cheung, and Cong Yan. 2020. Demonstration of Chestnut: An In-memory Data Layout Designer for Database Applications. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384712

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) while using relational databases for persistent data storage. The object-oriented programming paradigm makes it easy to develop applications: data to be persistently managed are organized into classes, each class is mapped to a relation, and each object instance becomes a tuple. Besides, by utilizing relational databases, such object-oriented database applications (OODAs) can leverage relational query optimization techniques to more efficiently manage application data.

In practice, however, OODA queries have different characteristics than analytical or transactional queries. Most importantly, OODAs define one-to-many and many-to-many relationship between classes as "has_one" or "has_many" (e.g., in a project management application, a `Project` "has_many" `Issues` and a `Issue` "has_one" `Project`), and use the nested data model rather than relational when processing the data in the application (e.g., an array of `Projects` where each `Project` object contains a nested array of `Issue` objects). The object queries also return results in such nested data model. Such model can easily create deep object hierarchy, while data stored in relational databases are maintained in a tabular format. Moving data across the database and application incurs serialization cost (i.e., converting materialized join results into nested objects), which can often take longer than the query execution itself.

The above aspects make OODAs challenging to optimize using standard query processing techniques. In Chestnut, we propose storing data in a customized, nested data layout when loading data from disk to memory, *in addition to* storing in tabular layout in the disk. To find the optimal data representation for each OODA, we have implemented Chestnut, an in-memory data layout and query plan generator for OODAs. Chestnut leverages recent advances in program analysis, symbolic execution, and solvers to automatically generate a custom data layout and query plans given application queries.

To use Chestnut, user provides as input OODA source code (written using a subset of the Ruby on Rails API) and a memory budget. It searches for the best in-memory layout that optimizes for overall query performance, and outputs C++ code that implements the data layout and query plans.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384712

This paper presents a demonstration for Chestnut. Our demo comes with three main features. First, it visualizes the data structures generated by Chestnut as well as the query plans to let users better understand the output of Chestnut. Furthermore, the visualization contains an animation showing how the data structures are populated from the tabular data in the database, as well as an animation showing how the query plan generated by Chestnut is executed at each step. Second, it comes with a backend system that runs the Chestnut-generated code, and users can see the Chestnut-generated query's performance and compare with the original query performance while running on relational databases. Third, the demo includes a few user interactions. It lets users change the application workload like the queries to be included and the query weight to understand how it affects the data layout, as well as the memory bound. It also lets users to partially design their own data layout. Users can create their own nested data layout via our drag-and-drop interface, and let Chestnut generate a query plan to see how the performance differs from the Chestnut-generated layout.

2 SYSTEM OVERVIEW

This section gives a background on OODAs, followed by a description of Chestnut architecture and its interaction with the demo application.

2.1 Object-Oriented Database Applications

OODAs are often structured in a three-tier manner: a client, an application server, and a backend database. When an application runs, the client issues an HTTP request to the application server. The server then generates object queries to retrieve the data from database. The object queries are translated by an object-relational mapping (ORM) frameworks like Ruby on Rails [3] into a SQL query to run on the backend, and the query result is deserialized into objects and nested objects to the application server. The server processes the result to render a webpage which is sent to the client.

Chestnut changes the above workflow by replacing the ORM and in-memory database running SQL query utilizing Chestnut-generated data structures and query plans that are customized for the input OODA.

2.2 Chestnut Overview

As input, Chestnut receives a list of queries that can potentially be invoked by the application and a memory bound. Chestnut then determines the best data layout and query plans for the given OODA queries. The best data layout can store data in a nested format similar to the nested data model used in OODAs. For example, in a project management application, `Issues` associated with each project can be stored as a nested array inside each `Project`, rather than as two separate relations connected via foreign keys. To find the best layout, Chestnut 1) enumerates different ways to nest classes used in a query, 2) enumerates indexes, including clustered and non-clustered index, to add on each nesting that are potentially used by the query (Figure 1(a)), and 3) searches for a query plan for every data layout where one layout includes one way to nest classes as well as the added indexes (Figure 1(b)). Along with enumeration, Chestnut estimates the memory cost for each data layout and the computation cost for each query plan. To find the best layout within a memory bound that optimizes the overall query time, Chestnut formulates this problem into an integer linear programming (ILP) problem (Figure 1(c)). It uses an external ILP solver to solve the ILP formulation and, once solved, generates C++ code for its chosen data layout and query plans. Our research paper [10] covers the technical details.

2.3 Demo Workflow

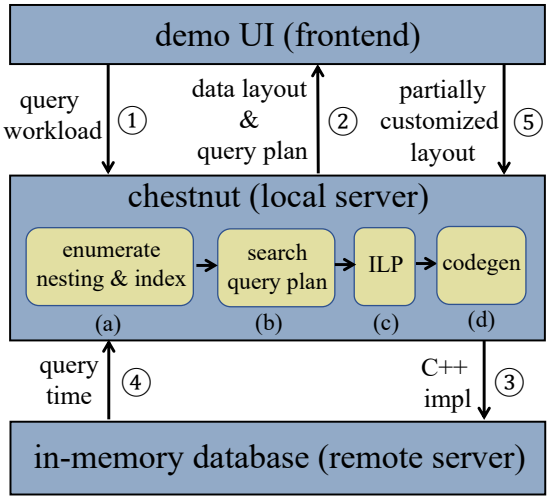

Figure 1: Demo architechture and workflow

This demo will allow the audience to visualize the hierarchical layouts and query plans that Chestnut generates, as well as comparing the performance gain it provides by automatic specialization.

The demo will be structured in the following way. An OODA will be hosted a local server along with Chestnut, with a backend in-memory database hosted on a remote server, as shown in Figure 1. The application server will generate a set of webpages to be rendered on the front end. Each webpage contains a set of webpages, each corresponds to a query that retrieves data to render for that webpage. The user can define a workload by choosing the set of webpages or changing their weight. The user also chooses a memory bound as a constraint to Chestnut. The workload and the memory bound is passed to the local server (Step ① in Figure 1). On the local server, Chestnut determines a data

layout and query plan for each query in the workload, and generates both an internal representation of the data layout and query plans to send to the front end ②, and the C++ code which is sent to the remote server ③. The front end visualizes the data layout and query plans in a web browser. The remote server compiles the CHESTNUT-generated C++ code, loads the application data into the data layout, and runs the query plans upon request. It runs the CHESTNUT-generated plans and the original query (including the SQL query and the serialization of query result). Both query times will be passed to the front end to show to the user ④.

Furthermore, users can partially design their own data layout via our demo's frontend. They can design how classes are nested using drag-and-drop interface. The nesting will be passed to CHESTNUT ⑤. CHESTNUT will add indexes on the user-defined nested layout and generates customized query plans. The layout and query plans are visualized as described above, and users can compare the performance between the self-designed and CHESTNUT-generated layouts.

3 DEMONSTRATION WALK-THROUGH

We now describe the different demonstration scenarios that the audience will experience through our demo.

3.1 Building A Workload

Our demo will come with two real-world open-source web applications written using Rails (described below) and allow users to customize these applications.

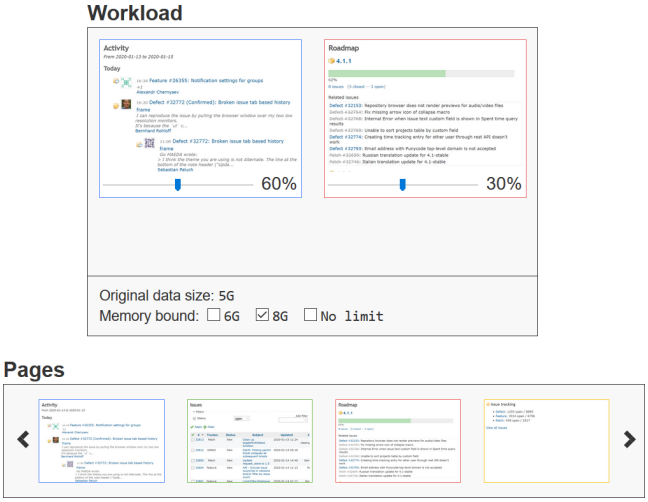

Figure 2: Example of a workload builder. Users drag and drop pages to add or remove them from the workload. Each page has a "weight" representing what percentage of traffic hits that page.

- *Kandan* [1], an online chatting application like slack. The abridged version is structured with three classes `User`, `Channel`, and `Activity`. We will select 8 webpages from the application rendering data like all the channels (with each channel containing activities), top recent activities, etc. Queries in this application contain simple predicates, but retrieve deep hierarchies of nested objects.
- *Redmine* [2] is a collaboration platform like GitHub. The abridged version is structured around multiple classes including `Project`, `Issue`, `Issue_status`, etc. This application contains 12 webpages which render data like projects with open issues, issues assigned to a particular user, etc. Queries in this application often involve multiple classes with complex predicates to retrieve the associated objects.

Users will first choose one of the applications. We will next show a collection of most frequently visited webpages as thumbnails. Users will build their application by choosing a set of webpages, as shown in Figure 2. The data shown in each component is retrieved by an object query. Users can click to view the corresponding object query that will be executed by each component when run in Ruby and the corresponding class models involved.

Besides composing webpages, users can also assign a weight to each webpage which indicates how frequently it is visited. CHESTNUT will prioritize optimizing for queries on higher-weighted webpages when generating data layouts, and users will be able to compare how different weights impact the generated data structures via our demo.

Meanwhile the user will see the original data size and then choose one of the provided memory bounds, as shown in Figure 2. The demo provides a few memory bounds that will make CHESTNUT generate different layouts, which helps the user understand the memory-performance tradeoff.

3.2 Layout and Query Plan Visualization

After designing a workload for CHESTNUT to analyze, users need to understand the data layout and query plans chosen. We will develop an interactive visualization system which shows in-memory data layouts and illustrates query plans while they execute.

An example visualization is shown in Figure 3, where each colored rectangular "block" represents an individual tuple (for tabular layout) or object (for CHESTNUT-generated layout). For hierarchical layouts, blocks will contain other nested blocks within them (which may contain more nested blocks, recursively). For instance, Projects `P` contain their nested Issues `I`, which in turn each may have an issue status `S`, as shown in Figure 3.

Our demo also shows how the data layout is constructed with the data stored in database with an animation. In the animation the record block from tables will move to its corresponding object in the in-memory layout.

Each data layout comes with a query plan. Our demo contains a visualization of the CHESTNUT generated query

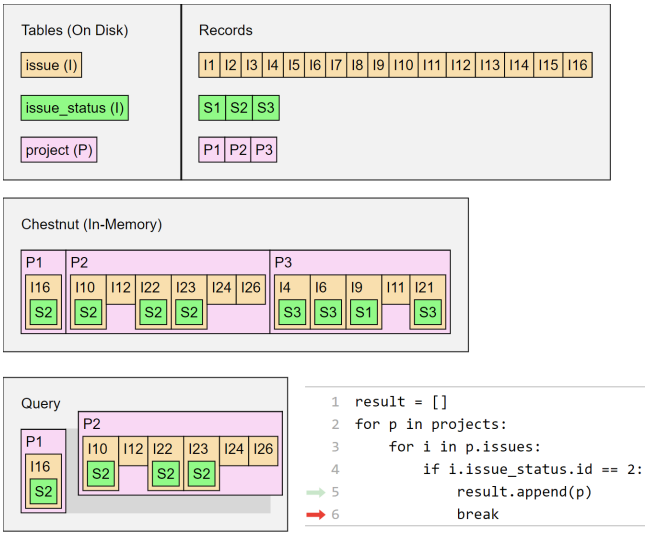

Figure 3: An example visualization. Records on the top represent disk storage. Middle shows the in-memory data layout, where `Projects` (`P`) contain nested `Issues` (`I`) with a nested `Issue_Status` (`S`). At the bottom, a simple query runs, returning projects with issues with status.

plans which is shown with the pseudo code and animation of the plan execution side-by-side, as shown in Figure 3. User will be able to step through the demo's animation, where each line of the code will be highlighted with data movement animated. In this and all the other animations, users will be able to pause and adjust the speed of animations to give them time to properly understand what is happening.

Finally, after seeing data structures and query plans, users can run the Chestnut-generated queries alongside equivalent SQL queries on large-scale datasets on a remote machine. The demo displays the timing results in real time, letting users observe Chestnut's speedup.

3.3 Interacting With Data Layouts

After visualizing Chestnut's chosen data layout, our demo will allow user to design their own layouts. This will provide a deeper understanding of how Chestnut's nested data structures work, and let users compare their designs against those generated by Chestnut.

Users will be able to drag and drop blocks in the Chestnut layout shown in Figure 3 within each other to design a new nesting. For instance, rather than `Projects` containing `Issues` containing `Issue_Statuses` (as in Figure 3), the user could assign `Issue_Statuses` as the top-level item containing issues which each contain projects. In this case, individual projects will exist multiple times in different issues, which will make queries selecting a set of projects slower.

Chestnut will then add indexes to the layout and generate the optimal query plan for the user-designed data layout. Users can then compare their design with the original design using the visualization system, and also see the differences in terms of memory usage and speed.

4 RELATED WORK

There is much prior work on specializing data structures, data layout, and physical design. In data structure design, Idreos et al. [9] customizes a key-value store that changes the elementary data structures according to the query pattern. Hyrise [8] and H2O [4] explore column store layout and decides storing columns together as group instead of individually, and ReCache [5] uses dynamic caching to store data in tabular and nested layout to accommodate heterogeneous data sources like CSV and JSON data. AutoAdmin [6, 7] finds best indexes and materialized views by trying different possibilities and asking the query optimizer for the cost. Unlike previous work, Chestnut co-designs data layout and the plan, and is able to explore a greater spectrum of data layouts which can better optimize OODA queries.

5 CONCLUSION

This paper describes the demonstration of Chestnut, an in-memory data layout and query plan generator for OODAs. Using our demo, users will be able to visualize how Chestnut specializes the data structures and query plans given an OODA. Furthermore, users can design their own specialized data layout and compare the performance of their designs against that generated by Chestnut.

6 ACKNOWLEDGEMENTS

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

REFERENCES

[1] Kandan, a chatting application. https://github.com/kandanapp/kandan.
[2] Redmine, a project management application. https://github.com/redmine/redmine.
[3] Ruby on rails, a ruby web application framework. https://rubyonrails.org/.
[4] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, pages 1103–1114, 2014.
[5] T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *PVLDB*, 11(3):324–337, 2017.
[6] N. Bruno and S. Chaudhuri. Constrained physical design tuning. *PVLDB*, 19(1):4–15, 2008.
[7] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *PVLDB*, pages 146–155, 1997.
[8] 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.
[9] 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.
[10] C. Yan and A. Cheung. Generating application-specific data layouts for in-memory databases. *PVLDB*, 12(11):1513–1525, 2019.