Revision Letter

We revised the article according to your suggestions and believe that, thanks to your feedback, we were able to significantly improve it. Sections 2 and 5 were substantially rewritten. We added Section 5.1 and Section 7 and rewrote the Abstract. Several other passages were revised and improved. All changes are highlighted in blue.

The key contribution of the article is a domain-specific IR that is designed for fast compilation of database workloads. We believe that the changes in the Abstract and in Section 1.2 help emphasize this key contribution.

In the added Section 7, we consider the tradeoff between generality and specialization for the design of IRs. Flounder IR chooses the specialization-side of this tradeoff and therefore has limited generality (e.g. limited portability to other hardware platforms). The added section discusses ways of increasing the generality of Flounder IR that may be able to retain its benefits.

Please find the detailed answers to the review comments in the following.

Reviewer 1

The paper argues that one of the reasons why they are faster than generic compilers is that the repetitive nature of query operators and their narrow scope allow for optimizations and focus on a few constructs rather than having to compile arbitrary code. To what extent is the mapping of, e.g., a join, a true compilation step as opposed to just instantiating a machine code template for the join? What applies to interpreted query execution, where the operators are generic and just invoked on different inputs, would seem to apply as well to operators in machine code. In 3.2 function calls are mentioned but the paper never explains how often they are used and the mechanism behind them (is there a catalog somewhere of available functions? If yes, how much actual compilation is happening as opposed to scripting a series of function calls?)

Response Patching machine code templates may be an interesting direction for achieving low compile times. However, such a technique is different from ours. Our approach achieves memory efficiency by sharing values in processor registers across multiple operators.

In a patch-based approach, the templates contain pre-defined uses of their values and different patches have to share values with each other. If the templates contain actual machine code, this means that they contain hard-coded registers. Every template has to clear its registers to allow them to be used by the next template. This requires materialization, which we are able to avoid with our approach. Alternatively if we use machine code with placeholder-registers, we

would return to a similar register allocation problem as addressed by our approach.

Regarding function calls, we use a similar approach as Hyper. We use IR code generation for data movement (e.g. reading a tuple from a hash table), and we use function calls for some laborious methods that are error-prone to implement via code generation (e.g. `like`).

It would be interesting to see some comment/analysis on the end-to-end steps of query compilation. The paper claims to be end-to-end but it only deals with the mapping plan → IR → machine code. The input is a query tree that, presumably, has already been optimized. Any chance to provide a rough idea of the relative cost of the complete sequence? SQL text string → query plan → optimized plan → IR → machine code? That would be a good way to frame the results. I see the point that query compilation is too long for short queries but, for complex queries (unlike what the paper claims), it is likely that the optimization step and running time make the compilation time irrelevant. The results of Figure 15 go along these lines but should be commented in more detail.

Response Query optimization is indeed an important step in the compilation process that comes at a cost. In fact, as shown by the Hyper-team, the optimization of query plans has a more dominant effect on query execution performance than the execution model itself.

However, a further analysis of the time spent during query optimization would have a limited contribution in this work and focus on an orthogonal aspect. On the one hand query optimization is a well-understood problem and commercial systems show strong optimization performance (e.g. answering full queries in few milliseconds). On the other hand ReSQL's query optimizer is not representative of such state-of-the-art query optimizers. The comparison in end-to-end performance with breakup in compilation and execution is the best estimate we can currently give. The results already show the importance of IR compilation times for several workloads.

There is quite a bit of work on hardware optimized operators (see, for instance, the papers by Balkesen et al. on radix hash join and sort merge joins in ICDE 13 and VLDB 13). These operators make heavy use of low level machine instructions (AVX, prefetching, etc.). It would seem that the approach in this paper would be a great way to implement such sophisticated algorithms. It would be good to comment on that, extending the discussion in 5.2. Along these lines, it would seem that the use of a prefetcher is something that could be indicated by the query optimizer rather than just generated in the compilation step. Are there opportunities to pass such hints (AVX, prefetching, data sizes, selectivities, etc.) down the pipeline to be used in the compilation step?

Response Thanks for these valuable ideas. In fact we are actively working on finding good ways to pass hints for more extensive hardware

optimizations to the IR. We improved Section 5.2 and now discuss different hardware architecture targets in Section 7.2.

It would be useful to see a discussion of any potential shortcomings of the proposed approach. The paper repeatedly claims that it is simpler and faster, which it proves in the experiments. But the paper does not prove or even discuss the generality of the approach and whether there are possibilities using LLVM that are not available using Flounder. For instance, LLVM is used to deal with heterogeneous hardware while Flounder is very much tied to x86 processors. Similarly, the paper only discusses a handful of operators and query types while the systems it compares to are capable of supporting a wider range of SQL constructs and queries. Some of it is just because Flounder has not been developed further but it would be good to show arguments that it can actually be extended and where its limits are.

Response With the goal to reduce compilation cost we designed our approach to be specialized. It is specifically made for database workloads and for one hardware architecture. This certainly is a limitation, but the approach allowed us to provide insight on the feasibility of query compilation with low compilation times. The degree to which it is possible to increase aspects of generality while retaining similar performance is a particularly interesting open question. We now discuss this aspect in Section 7.

I was puzzled by the "?" in Figure 1. It is not clear what it means or what it represents and the text does not explain it. I suggest to either clarify, give it a name, or remove it.

Response We added an explanation for the bar with the question mark in Section 1.1.

I would add in the introduction a line or two quantifying the results (5.5 speedup). It is in the abstract but the intro only has a generic description of the results. It would also be useful to list there already what systems are used as baseline for comparison.

Response We added a statement about the speedups in Section 1.2.

In Section 1.2, I would remove the explanation of the added contents over the DAMON paper. It is useful for reviewing but nor needed for the final version.

Response We removed the comments on the additions for better readability.

I am not a great fan of having an outline for the next sections every few paragraphs. It is not needed and it looks like the text is there to make the paper longer. There is already an outline in the introduction, I would suggest to

remove the repetitive outlines added in several sections (e.g., at the end of 2.2).

Response We removed the outline at the end of Section 2.2 and revised the introduction of Section 4 in this regard.

For the final version, it would be nice to correct paragraphs that end with one or two words in the last line. There are quite a few.

Response We fixed several paragraphs that ended with one or two words.

Some of the code is labeled as a figure, other is inlined in the text and without label. I suggest to be consistent and label all of them as Figures. Also, in 2.3, I would suggest to make the example a figure rather than having it floating on the side of the text. It does not look nice and it is not needed as there is space for it.

Response In the revised version of the article, we changed several previously inlined pieces into Figures. The changes are for Figure 2, Figure 5, and Figure 7 To avoid an over-crowding with figures we kept several shorter code pieces in line.

Reviewer 2

Regarding Fig. 3: I am not familiar with the produce/consume model of [29]. I guess that because of that, I missed the outer-loop of the Build-Side where all tuples of the build-side are inserted into the hash table. I also could not follow the meaning of a1 vs. Attributes in this figure. I propose to expand a bit more on explaining the Figure.

Response We carefully revised Figure 3 (now Figure 2) and its description to not rely as much on previous knowledge of produce/consume. The first paragraph now differentiates more clearly between the build and probe pipelines. In the description of the probe pipeline we clarified that the scan operator emitter is responsible for the outer loop. In the explanation of the pseudocode, we now use `createHashtable(..)` as example for functionality that is executed once during translation and `ht_ins(..)` as example for functionality that is placed in the compiled code and may be executed executed repetitively (e.g. via preceding scan operators that emit a surrounding scan loop). We added a comment in the pseudocode to clarify the role of a_l.

Section 2.3 contains well-known compilation techniques. In my opinion it does not add to the depth of the paper, and therefore can be omitted. By contrast,

Section 3.1 does contain a new technique of scoping the use of virtual register in SQL processing loops. Typo in Section 2.3: The constant 0.23 is replaced later by the constant 0.34.

Response We shortened this part to contain only two aspects with direct relevance to Flounder IR. We also condensed the descriptions and fixed the typos regarding the decimal constant.

Figure 10: You need to solve the editorial/visual problem of how to annotate the X-axis. There are 2 different scales: 1 for projection and 1 for join. But the figure names none.

Response We revised the plot and now name both scales. We also changed the order of the referenced graphs in the legend to improve readability.

From my perspective, the key contribution of the paper is the demonstration of a domain-specific JIT compiler (for the specific domain of SQL), as opposed to the general-purpose LLVM. For example, the special and simple register allocation scheme benefits from the domain specificity. I would recommend emphaizing this point. Section 1.1 indeed introduces this contribution, but it deserves further elaboration (maybe ast future work, for other domains such as AI/ML...).

Response Thanks for these valuable suggestions. We revised the abstract and the introduction to improve the presentation of Flounder IR as a domain-specific compilation technique. In the revised paper, we also include Section 5.1 and Section 7.1, which relate to the topic. Section 5.1 discusses additional database-specific knowledge, which can potentially improve our approach. Section 7.1 discusses the combination of database processing with other domains.

Reviewer 3

As a general comment, given the focus on latency it was a little surprising to see the evaluation with OLAP workloads in a tiny database in Section 6.6. These workloads have long-running queries that can tolerate a few seconds of delay due to compilation. It may have been more interesting to evaluate Flounder IR and ReSQL with OLTP workloads, where compilation would add an unacceptable latency to every transaction to the point where compilation performance alone determines the peak throughput of the system.

Response The suggestion to apply our approach to transactional workloads makes sense. Currently, however, ReSQL only has a very basic approach to concurrency. Thefore such workloads would be dominated

by synchronization, leading to an uninteresting comparison. A future improved concurrency protocol would make this a meaningful analysis.

How are NULLs handled in ReSQL and Flounder IR? The instruction path length grows considerably when NULLs are involved due to the use of binary logic in the ISA. Flounder IR has an opportunity to encode ternary logic directly in the IR. Some discussion on this would strengthen the paper.

Response This is an interesting comment. Currently expression translators have to implement handling of `NULL` values. For the revised paper, the added Section 5.1 discusses opportunities for improving the handling of `NULL` values.

Please clarify the relationship of the decimal type used in the example in Section 2.3 to the SQL exact numeric types (numeric, decimal) and the approximate numeric types (float, real). In particular, clarify if ReSQL supports both exact and approximate numeric types, or if approximate numeric types are converted to exact types? Also, clarify how are exact decimals wider than the register size handled.

Response We clarified in Section 2.3 that ReSQL uses 64 bit integers to represent decimals. Currently ReSQL only supports exact numeric types.

The prefetching optimization in 5.2 aims to match the granularity of prefetching with the granularity of access by tracking cacheline accesses and using loop unrolling. This realization comes later in the section and is a little underwhelming—I was reading eagerly for an interesting way to optimize the prefetching distance which is a very brittle optimization in practice. Please state the nature of the optimization upfront, instead of at the end of the section. Some discussion on if/how a database-specific IR can help with tuning the prefetch distance would also be welcome.

Response Thanks for this comment. After re-reading we realize the issue. We revised the Section to be more up-front about the nature of the optimization. We also added a discussion on optimizing the prefetching distance.

The 100MB database used in Section 6.6 appears to be outside the specs of TPC-H, where a scale factor of 1 produces a 1GB database. How was the 100MB database generated exactly?

Response We used the 100 MB database to challenge Flounder with shorter processing times. We generated the data with `dbgen -vf -s0.1`. While this is not a standard size for the TPC-H benchmark, the generated data appeared sound under visual inspection.