We thank the reviewers for their valuable feedback. We provide detailed responses to each reviewer's questions below. The concerns are mainly about missing details and some aspects of our evaluation. We hope that the reviewers will agree that the missing details can be easily clarified in a revision. Moreover, while we oriented our evaluation along the lines of previous work, specifically Correlation Maps [1]) and Hermit [4]), which used similar B-Tree baselines, we have other baselines / workloads readily available and would be happy to add them in a revision.

Reviewer #1

O1. There are indeed many variants and optimizations of B-Trees used in practice. For Cortex, we actually use a cache-optimized index with bitmap index scans and late materialization (see below), but we acknowledge this is only one possible variant. However, when we wrote the paper, we believed that our comparison to B-Trees was sufficient because:

1. It is consistent with prior work discussing correlations ([4, 1]). CM uses a B-Tree with a bitmap index scan, while Hermit simply uses a cache-optimized B-Tree.
2. Regarding the evaluation, our main goal was to understand whether Cortex's algorithm to select outliers based on query performance (given a dataset and host index) can improve performance-space tradeoffs over previous attempts to leverage correlations and existing indexes. We believe that our evaluation demonstrates this tradeoff effectively.

However, we do agree that additional baselines would help to better understand the benefits of Cortex and other similar approaches like CM and Hermit. We would be more than happy to include them as we do believe it provides additional value. That said, we believe that many of these additional optimizations and alternative baselines will not qualitatively change the results as outlined below:

1. **Late materialization**: Our implementation uses this optimization, and applies it to all indexes, not just the B-Tree. We access only filtered columns to determine the matching records (in the code supplement, `cxx/src/query_engine.hpp:55`) and only then materialize the matching tuples (`cxx/src/query_engine.hpp:60`). We will make this more explicit in a revision.
2. **Block-skipping**: The number of blocks which can be skipped depends heavily on how the data is sorted between blocks: sorting on uncorrelated column(s) will likely not result in substantial pruning, while sorting on highly correlated column(s) is the approach used by CM. Therefore, even though it reduces space overhead, it is unlikely this approach will outperform the CM baseline.
3. **Bitmap index scan**: This type of scan is also used by our implementation and is applied to all indexes. As shown in Figure 1, we operate entirely using indexes and bitmaps (represented by rectangles with green to denote set bits) until the very end, when we have to access the column data to materialize the results. Our implementation uses a bitmap, or selection vector, implementation (like [1]) optimized for ranges and low selectivities like those in our workloads. For reference, the intersection of these vectors happens at `cxx/src/composite_index.hpp:126,189`. We will explicitly label this technique in a revision.
4. **Covering indexes**: These occupy a large space overhead since they duplicate columns within the index - like materialized views. While potentially leading to better performance, this type of index would lie even farther to the right of the B-Tree in Figures 6 - 8. Therefore, we did not consider it as a comparable baseline in our paper.
5. **Composite key indexes**: to some extent these can be considered multi-dimensional indexes that prioritize keys in a given order. In fact, a composite key is just a special case of a Flood index (with partitions along every attribute value). Hence, we already considered it covered. However, we would be happy to include a composite key as a explicit baseline in a revision.

O2. Thank you for pointing out missing details. Here are the answers to your questions, in order:

- The dictionary is order preserving so that range filters on raw values translate to range filters on encoded values.
- It is built by sorting the values for that field and assigning them increasing values. Since the main results do not include inserts, this was enough for our purposes.
- Index structures are built on encoded keys.
- Bitpacking is not used on indexes, only on the underlying data. Our workload focuses on non-string keys and for which prefix compression of indexes is not as beneficial.
- The 1-D index is not a B-Tree; it performs a binary search within the column. The clustered key index is only used to map the key of a record to its physical location in storage, useful for md-indexes (see Figure 1).
- Why does Cortex not support strings / floats? Cortex supports any datatype that can be ordered. The *column-store implementation* we use is the only part that does not support variable-length strings. It does support floats, but we actually found operations on integers were faster and resulted in slightly better performance overall, which is why we convert them to integers; this is equivalent to how SQL server stores fixed-precision decimals [3]. These restrictions are also consistent with the evaluations in prior work (cf. [2, 4, 1]).

We will add these clarifications to the evaluation section in a revision.

O3/O4. Note that Figure 12 already evaluates Cortex against an Octree baseline, which is also a host index. An Octree *is* a traditional md-index, since it is just a quad-tree in higher dimensions. We also evaluated against a Flood baseline (below), but since it looked similar, we did not include it in the paper. We would be happy to include this or explicitly mention the similarity in a revision.

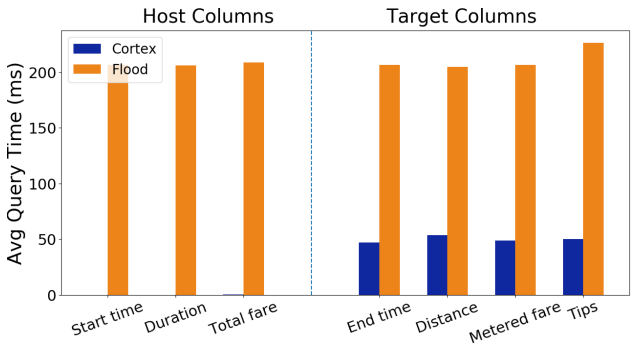

Figure 1: Cortex w/ Flood host evaluated against a baseline where Flood is used to index all columns. Flood performs poorly since its uniform grid structure does not let it adapt to data correlations. These data lead to a similar conclusion as Figure 12 from the paper.

As outlined in O1, we oriented our evaluation along the lines of prior work, namely CM and Hermit (although our datasets are orders of magnitude larger). These works did not compare against multi-dimensional indexes, and they almost exclusively use single-predicate queries. That said, we agree with the reviewer that it would be interesting to include them as baselines in Figure 6-8 and beyond. Therefore, for a revision, we will not only include the other baselines but also add micro-experiments with multi-predicate queries.

Reviewer #2

O1. Please find a detailed discussion of how Cortex scales to many columns in Section 5.5: Scalability.

Reviewer #3

O1: The underlying column-store implementation we use does not handle variable-length strings, since this added complexity to the implementation without providing additional insights on Cortex on top of what numeric fields were able to provide. This restriction is also found in other similar evaluations (c.f. Flood [2]).

O2: We obtained $\beta = 17.88$ through the process described in Section 4.3. The R^2 value was the strength of our linear model's fit to the actual performance data.

O3: We only used the B-Tree as a baseline, because our other baselines are multi-dimensional indexes, which fall prey to the curse of dimensionality: after only a small number of columns, these indexes quickly become as slow as a full table scan. For example, the figure below shows the curse of dimensionality for an Octree:

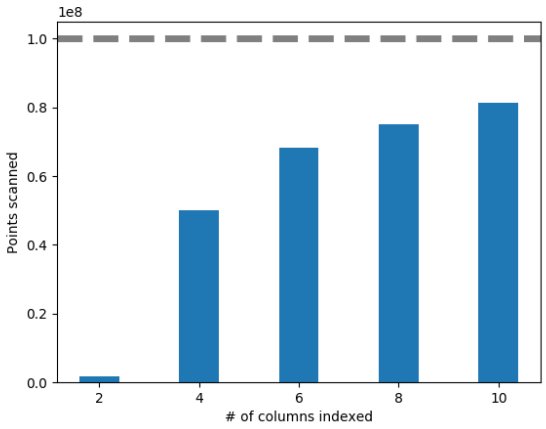

Figure 2: The gray dotted line is the size of the dataset, and the bars indicate how many points are scanned by the octree and quickly approaches a full scan as the number of columns increases. This experiment used a synthetic dataset with linear correlations.

This result is corroborated by prior work as well (see Figure 13 in [2]). Therefore we didn't think these results would be a valuable addition to the scalability experiments. If the reviewer finds them useful, we would be happy to include them in a revision.

References

[1] H. Kimura, G. Huo, A. Rasin, S. B. Zdonik, and S. R. Madden. Correlation maps: A compressed access method for exploiting soft functional dependencies. In *VLDB*. ACM, 2009.

[2] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 International Conference on Management of Data*, SIGMOD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[3] R. West. How SQL Server stores data types: integers and decimals. https://bornsql.ca/blog/how-sql-server-stores-data-types-integers-and-decimals/.

[4] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1223–1240, New York, NY, USA, 2019. Association for Computing Machinery.