CS 764: Fall 2018. Project Ideas.

This document is not meant be exhaustive. Please think about these project ideas, but don't be afraid to suggest a project significantly different from those below. Be creative – this is a chance to explore data management techniques and issues in depth.

1. File systems and database systems has a lot of similarities and complementary aspects. Can we bring them closer? Do this for one aspect, i.e. search. Extend the UNIX file system by adding a new call to allow a user to describe some structure for the data (e.g. a protobuf) and then allow search on that (e.g. by implicitly building in a B+tree underneath the covers in the filesystem).

   Explore how far you can take this idea to bring filesystems and database closer. See earlier work on this: Margo I. Seltzer: Berkeley DB: A Retrospective. IEEE Data Eng. Bull. 30(3): 21-28 (2007)

2. There is significant work on hash join algorithms for main-memory, multi-core, multi-socket settings for both hash and sort-based algorithms. See:
   - Stefan Schuh, Xiao Chen, Jens Dittrich: An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. SIGMOD Conference 2016: 1961-1976
   - Spyros Blanas, Yinan Li, Jignesh M. Patel: Design and evaluation of main memory hash join algorithms for multi-core CPUs. SIGMOD Conference 2011: 37-48
   - Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, Pradeep Dubey: Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. PVLDB 2(2): 1378-1389 (2009)

   But, there isn't a clear answer about the winners between sort-based or hash-based methods when considering the range of input key sizes, tuple sizes, and impact of aspects such as whether result tuples are materialized or not. Investigate this, and try to find a clear answer.

3. Presume that you have a storage system with a key/value atomic write primitive. Further assume that the writes are atomic at arbitrary size. Can you use this mechanism to simplify recovery – i.e. get rid of logging in most/all cases? Build a prototype storage system to explore this idea along the spectrum from two writers modifying the same byte of data in a page to up to a large bulk load scenario, and compare the performance against a traditional WAL scheme in both the steady state (normal transaction commits/aborts) and in crash recovery for a variety of read/write workload mixtures.

4. There is a new class of data applications called Document Stores. More often than not, document store data platforms (e.g. MongoDB) are built independent of relational systems. Is there a good reason for this, or can relational systems subsume document stores? Explore this by building on the ideas presented in:

- Craig Chasseur, Yinan Li, Jignesh M. Patel: Enabling JSON Document Stores in Relational Systems. WebDB 2013: 1-6

Build an ARGO mapping layer on top of PostgreSQL. Compare with MongoDB. What conclusions can you draw from this about the need for specialized systems.

5. To support arbitrary search on string, a powerful data structure/index to use is called a suffix tree/suffix array. Build a suffix tree/array index on string attributes in Postgres or other open source DB and explore its use for regular expression and substring search.

6. Databases for tracking customer sales tend to suffer from lock contention as they scale up the number of concurrent new order creation. When a new order is created, a sequential order number is assigned to it. When the record is inserted, the index on the order number must be modified with the new order number. Because the new order number is the largest such number, it always goes to the rightmost leaf page of the B-tree index. Thus, that page is almost always locked for writes, and when splits occur it cascades into a tree rebalance at a high cost to concurrency. Design strategies for dealing with this problem automatically and implement them to compare their concurrency/throughput behavior.

7. Consider unifying query optimization and query processing. Think of the enumeration aspect of query optimization as a way to "compute" all possible combination of a way to do a (sequence of) joins. This can likely be expressed as a join on a well-constructed table that is created just for this purpose. Thus, one can enumerate the joins using an actual join operation. How far can you push this idea?

8. Non-volatile memory is poised to transform the memory hierarchy of compute systems over the next decade. It promises to offer byte-addressable permanent storage an order of magnitude larger than RAM at latencies several orders of magnitude less than flash/SSDs. Choose an aspect of a DBMS that interacts with storage (e.g. sorts/joins, as found in http://www.vldb.org/pvldb/vol7/p413-viglas.pdf) and implement against a memory-resident storage simulator (e.g all writes to this memory have appropriate delays inserted to approximate NVM). Explore the tradeoffs in implementing new algorithms for NVM – for example, should such storage be used for intermediate/work tables or reserved for permanent/user data?

9. One of the challenges of building parallel database systems is how to avoid operations that require all data to be examined in one place. One example is Full Outer Join – it is not possible to implement this correctly without seeing all possible matches for a tuple from the other relation. Perform an analysis of ANSI SQL constructs (especially modern SQL constructs like windowed analytic functions as used in TPC-DS) in the context of parallel database implementation and characterize each according to techniques for parallel implementation. What techniques should the optimizer and execution engine use to keep the computation parallel as long as possible?