# High Frequency Data and Raw Binary Databases

# We want to use high frequency data

- We want to run a backtest - a piece of code that evaluates a particular trading strategy using historical data.
- In a backtest, historical data is in sequence - from the oldest to the newest.
- If we are using the right framework, we read through this data one time stamp at a time, deciding when we want to buy or sell.
- We can make our decisions based only on the data we've already read, not the data remaining in the rest of our data set.

# Back-tests should be realistic

- Our backtest should be realistic, so we use an impact model to penalize large trades or rapid rates of trading.
- We also use a leverage model to force ourselves to stay within the credit constraints of our prime brokers. Our leverage model also charges us for borrowing money for our strategy.
  - Hedge funds have an average leverage of 3.5 to 1, so most of the money that they trade with is borrowed from their prime brokers, and for this they pay interest, and agree to stay within specified leverage limits.
  - This leverage and the non-linearities of having to reduce leverage to stay with our leverage constraints necessitate accurate simulations of trading.
    - For example, a panic liquidation of large positions – an event that may be required by a prime broker – has additional market impact

# The problems of using high frequency data

The data we typically want to read for the above task is in sequence, but has gaps that make it difficult to work with. For example, consider the following records:

Quotes

ts = 9:30:05, bs = 200, bp = $13.01, as = 100, ap = $13.03

ts = 9:30:08, bs = 200, bp = $13.01, as = 100, ap = $13.03

Trades

ts = 9:30:01, s = 500, p = $13.05

ts = 9:30:06, s = 500, p = $13.05

ts = 9:30:11, s = 500, p = $13.05

# Option 1 – Merge data sets without changing record structures

- We write the binary records to a new flat file just as they are structured above.
  - ts = 9:30:01, s = 500, p = $13.05
  - ts = 9:30:05, bs = 200, bp = $13.01, as = 100, ap = $13.03
  - ts = 9:30:06, s = 500, p = $13.05
  - ts = 9:30:08, bs = 200, bp = $13.01, as = 100, ap = $13.03
  - ts = 9:30:11, s = 500, p = $13.05
- This database is hard to read because it is not uniform. Its records don't all have the same number of bytes.
- We must determine on the fly what type of record we're processing, a quote or a trade, and this creates overhead.
- The quotes and trades now appear in two places - in their own files, and in the merged file - so every change to either quotes or trades results in a rewrite of two files instead of one.

# Option 2 – Join records

- We join the quote and trade records to create a longer record using the last available quote or trade.
  - ts = 9:30:01, s = 500, p = $13.05, ts = NULL, bs = NULL, bp = NULL, as = NULL, ap = NULL
  - ts = 9:30:01, s = 500, p = $13.05,  ts = 9:30:05, bs = 200, bp = $13.01, as = 100, ap = $13.03
  - ts = 9:30:06, s = 500, p = $13.05,  ts = 9:30:05, bs = 200, bp = $13.01, as = 100, ap = $13.03
  - ts = 9:30:06, s = 500, p = $13.05,  ts = 9:30:08, bs = 200, bp = $13.01, as = 100, ap = $13.03
  - ts = 9:30:11, s = 500, p = $13.05,  ts = 9:30:08, bs = 200, bp = $13.01, as = 100, ap = $13.03
- If unavailable for the current time stamp, the last available quote or trade record is replicated in the current record. The record time stamp is the greater of the time stamps of the trade record and the quote record.
- The database is now of uniform record size, but this approach leads to a lot of data replication and makes the database much larger.

# Joins are really bad when working with both low and high frequency data

- Suppose we were working with both prices and earnings data.
- The earnings statement comes out four times per year but has many, many fields.
- Suppose it has 30 fields.
- If we join 30 fields of earnings data with three fields of price data, we are bulking up our price database by a factor of 11.
- Almost all of the earnings data is replicated millions of times!
- The biggest bottleneck in our back-test is data read/writes, so this is a sure fire way to slow our back-test to a crawl.

# Option 3 – Read data sets separately but in order

- This is the most efficient option for reading flat sequential binary data – the kind that is typically used in back-tests.
- In this approach, we never combine different record formats.
- For example, suppose we are trying to work with both quotes and trades.
- All of the quotes are in one file and all of the trades are in another file. (If we were also working with earnings data, it would be in a third file.)
- We juggle our reading of the files by reading all files up to a certain time stamp, processing this data, and moving on.

# Reading logic

- We look to see which file has the lowest time stamp – in other words, which file should be read next. We can do this simply by sorting a list of active file handles – one per file – by the time stamp of the next record.
- Suppose the quotes file has the lowest time stamp. We save the reference time stamp – refTS - for this quotes file.
- We now read records from this quotes file until we encounter the first record with a time stamp that is greater than refTS.
- If the next record in the trades file also has a time stamp equal to refTS, we read the trades file until we encounter the first record with a time stamp greater than refTS.
- When we have read records from all of the files that have time stamps that are equal to refTS, we stop and call some code to allow all of these newly read records – from all files – to be processed.
- We repeat the whole process until we have no more data to process.