BMLL Data Feed

Level 3

Date: 2025-11-18
Version: 2.10

1.1. Introduction

An order-by-order view of the limit order book. Every single update — insert, update, cancel, execute — is captured, giving an in-depth understanding of historical market activity.

Use this dataset for microstructure research, execution-quality analysis, or rebuilding the full book at any point in time.

1.2. Dataset schema

Basic Types

Type	Content	Text representation
BOOLEAN	A True or False boolean value.	1,0
INT32	A 32 bit signed integer.	12345
INT64	A 64 bit signed integer.	12345
DOUBLE	IEEE 64-bit floating point value.	123.4567
BYTE_ARRAY	An arbitrarily long byte array.	"VODAFONE GROUP"
NUMBER	A number with defined precision (total digits allowed) and scale (number of digits allowed to the right of the decimal point). Unless otherwise specified, the default is precision 38 and scale 0.	12345
VARCHAR	Variable length string up to 500 characters. May contain quotes and special characters. All text will be utf8-encoded unless indicated differently. Ticker and ISO codes should be ASCII compatible.	"VODAFONE GROUP"
TIMESTAMP_NTZ	Nanoseconds since 1970 in UTC. Timestamp will be represented in ISO format in text format.	20220302-11:23.57.12345789
FLOAT	IEEE 64-bit floating point value.	123.4567
DATE	A calendar date consisting of a year, month, and day, without including any time or time zone components.	2025-01-01

T: +44 (0)20 3828 9000
E: info@bmlltech.com
W: www.bmlltech.com
Copyright © 2026 BMLL Technologies. All rights reserved.

1.3. Schema

One row corresponds to a single order book event (insert, update, or remove) at a specific point in time for a specific instrument on a specific venue.

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
MIC	MIC	VARCHAR	BYTE_ARRAY	The Market Identifier Code (MIC) is used to uniquely identify the exchange which is reporting market activity.
TICKER	Ticker	VARCHAR	BYTE_ARRAY	BMLL-format ticker for a listing on its trading venue. This is the customer-facing Ticker used in API, Snowflake, and Databricks materialisations.
EXCHANGE_TICKER	ExchangeTicker	VARCHAR	BYTE_ARRAY	BMLL-format venue ticker carried under the ExchangeTicker physical name in parquet/Revelate parquet schemas. API, Snowflake, and Databricks materialisations should expose this semantic as Ticker.

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
LISTING_ID	ListingId	NUMBER	INT64	The BMLL ListingId is used to identify a specific instrument on a market venue (trading venue or reporting venue). BMLL listings identify uniquely every ticker on the venue and remain attached to the instrument even when a ticker change occurs — though changes of primary instrument identifier (such as ISIN changes) can cause a new listing_id to be issued.
INSTRUMENT_ID	InstrumentId	NUMBER	INT64	The BMLL identifier of the BMLL instrument. BMLL instruments are designed to connect together listings of a designated underlying asset when they are issued with similar condition (i.e. same quoting currency, geographical, legislation region, primary reference venue).

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
TRADE_DATE	TradeDate	DATE	INT32	Date associated with the dataset row. Product-level smell records document any legacy product where this date is derived from publication time rather than the venue trade timestamp.
TIMESTAMP_NANOSECONDS	TimestampNanoseconds	NUMBER	INT64	This column contains a representation of the time of the update as the number of nanoseconds since 1970-01-01 00:00:00.0 UTC.
RECEIVE_TIMESTAMP	ReceiveTimestamp	TIMESTAMP_NTZ	INT64	BMLL infrastructure receive timestamp. Was referenced from asset.equity.l3's logical_schema with no backing column definition until 2026-09-08 (confirmed live via bmll2.get_market_data(schema='Equity', table_name='l3')).
EVENT_NO	EventNo	NUMBER	INT64	Synthetic event id allowing to group actions that occur atomically on the limit order book and during which no other order may have interacted with the book
SIDE	Side	NUMBER	BYTE_ARRAY	Side on which the order is resting

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
LOB_ACTION	LobAction	NUMBER	BYTE_ARRAY	The type of the instruction as relevant to rebuilding the limit order book state:
ORDER_TYPE	OrderType	VARCHAR	BYTE_ARRAY	Order type: 0 = Unknown; 1 = Limit; 2 = Market.
PRICE	Price	FLOAT	DOUBLE	price at which the order occurred after the event
SIZE	Size	FLOAT	DOUBLE	size of the order on the book after the event
ORDER_ID	OrderId	VARCHAR	BYTE_ARRAY	Order ID currently assigned to the order to which the instruction refers, as provided by the exchange. This is always the ID after this instruction is applied. For a lob_action of remove (or unknown), this is an empty string, ''.
OLD_PRICE	OldPrice	FLOAT	DOUBLE	Price of the order the instruction refers to before it is applied. For a lob_action of remove, update or skip, this is the old price. For insert (or unknown) this is Nan.
OLD_SIZE	OldSize	NUMBER	INT64	Size (quantity) of the order the instruction refers to before it is applied. For a lob_action of remove, update or skip, this is the old size. For insert (or unknown) this is 0.

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
OLD_ORDER_ID	OldOrderId	VARCHAR	BYTE_ARRAY	Order ID the instruction refers to as provided by the exchange before this instruction is applied. For a lob_action of insert (or unknown), this is an empty string, ''.
ORIGINAL_ORDER_ID	OriginalOrderId	VARCHAR	BYTE_ARRAY	Order ID originally assigned to the order to which instruction refers. In some cases order_id can change in the order's lifetime; this is always the first ID assigned to the order.
ORDER_EXECUTED	OrderExecuted	BOOLEAN	BOOLEAN	True when the instruction was an execution.
EXECUTION_PRICE	ExecutionPrice	FLOAT	DOUBLE	Execution price when the instruction is an execution. Equals the price except for market orders, and in special circumstances (see the relevant dataset page). When order_executed is False, this is NaN.
EXECUTION_SIZE	ExecutionSize	NUMBER	INT64	Execution size when the instruction is an execution. Equals old_size - size when order_executed is True, and 0 otherwise.

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
TRADE_ID	TradeId	VARCHAR	BYTE_ARRAY	When available, the Trade ID as provided by the exchange. The ID associated with an execution can be used to link to the Trades Table described below. Each value corresponds to a unique row in the trades table, if provided.
PRICE_LEVEL	PriceLevel	NUMBER	INT64	The level of the order after the instruction (1 = best price) where applicable. For a lob_action of remove, this is the price level of the removed order if the level still exists, or 0 otherwise. For lob_action of unknown this is 0.
POS_CHANGE	PosChange	VARCHAR	BYTE_ARRAY	True if the order's queue priority position changed as a result of this update.
OLD_PRICE_LEVEL	OldPriceLevel	NUMBER	INT64	The level of the order before the instruction (1 = best price) where applicable. For a lob_action of insert, this is the price level of the inserted order if the level previously existed, or 0 otherwise. For lob_action of unknown this is 0.

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
SIZE_AHEAD	SizeAhead	NUMBER	INT64	For lob_action = insert or update, the total size (quantity) of the orders ahead of the given order in the priority queue, after the event. For lob_action = remove, this is -1.
ORDERS_AHEAD	OrdersAhead	NUMBER	INT64	Number of orders ahead
END_OF_EVENT	EndOfEvent	BOOLEAN	BOOLEAN	Many rows (instructions, i.e. changes to orders) may occur for one event (i.e. transition from one valid order book state to another). This flag is true if this is the last instruction in the same event (see above).
MARKET_STATE	MarketState	VARCHAR	BYTE_ARRAY	Standardised market state information. See below for description of possible market states.
PRINTABLE	Printable	BOOLEAN	BOOLEAN	Printable value as computed by BMLL parser allowing to filter out duplicate (non-printable) trade reports. These can be synthetic trade report, or trade reports describing a same execution from the view of the counterparty.

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
DEL_ORDER_INDEX	DelOrderIndex	NUMBER	INT64	Zero-based index of the order to remove from the book array when reconstructing as an ordered list. -1 if not applicable.
ADD_ORDER_INDEX	AddOrderIndex	NUMBER	INT64	Zero-based index at which to insert the updated order into the book array. -1 if not applicable.
MPID_ATTRIBUTION	MPIDAttribution	VARCHAR	BYTE_ARRAY	Market Participant ID (MPID) of the firm attributed to this order or trade event, as reported by the exchange.
ORIGINAL_EXCHANGE_MESSAGE	OriginalExchangeMessage	VARCHAR	BYTE_ARRAY	Raw exchange message string, provided for diagnostic use.
EXCHANGE_SEQUENCE_NO	ExchangeSequenceNo	NUMBER	INT64	The sequence number as provided by the exchange. Note that some exchange may not provide a sequence number in this case the ExchangeSequenceNo will always be 0.
BMLL_SEQUENCE_SOURCE	BMLLSequenceSource	NUMBER	INT64	An integer used to identify the upstream feed from which the sequence numbers are generated. This is designed to avoid the creation of non-existent order between events that come from different message streams

Field Name (Snowflake / Databricks)	Field Name (Parquet)	Type (Snowflake / Databricks)	Type (Parquet)	Description
BMLL_SEQUENCE_NO	BMLLSequenceNo	NUMBER	INT64	The sequence no of the trade in the upstream feed as inferred by BMLL - this sequence number will respect the order of messages in upstream protocol and will be populated even if the upstream venue does not provide a sequence number - where the source of the feed is indicated by BMLLSequenceSource.

1.4. Delivery availability

Level 3 is available through the BMLL Data API at supported venues, subject to your dataset and venue entitlements. Native Snowflake table delivery requires a special commercial agreement. This requirement does not make API access an on-demand product.

1.5. Frequently Asked Questions

We can notice some orders that execute although they were not at the top of the book — is this expected?

Many venues have a special type of orders called Discretionary Orders. These orders will be sitting on a book at a predefined price but may have additional instructions, which can correspond to a second hidden limit against which the order may go aggressive. These instructions will often cause the order to sit passively on the book and, then, to execute before those that were before other orders of the order book, they can generally be identified by the fact that the execution price was ultimately different from the price at which they were sitting.

Another situation that can occur is that a locked NBBO quote protection may prevent the immediate of execution of orders on venue (orders flagged as Do Not Route (DNR) will stay blocked if the NBBO is crossed), and as NBBO unlocks, and the book evolves the order the execution that was previously blocked by a locked NBBO, orders may

adapt their price, matching may become acceptable, and matched execution may then occur though book that has evolved — but where order previously present on the book may benefit from specific rules allowing them to jump back ahead.

We can see some orders increasing their size and keeping their position in the queue — how is it possible?

In US markets, Reg NMS and best-execution obligations have technical implications that are important to understand when looking at limit order book data: An order or a part of an order that was sitting on the book may need to be routed from one venue to another venue to ensure best-exchange based on protected quotes. However, as routing and information dissemination inevitably take time to be transferred between trading venues, it can sometimes happen that, when the routed order arrives on the venue, the quantity is not available anymore as other venues may have also routed orders to the most competitive venue. When this happens, the quantity will be routed back and reinserted back in the limit order book of the original venue — when this quantity comes back, it is reinserted where it was (without losing its priority). This feature relies on the PosChange which is populated based on exchange data for BATS, BATY, EDGX, EDGA and EPRL.

We can see orders being added during trading halts — is this expected?

Yes, this can occur for multiple reasons, though many venues will reject order entry during trading halt, this is not always the case and some venues will still accept order change during halt. Also, even if order entry is closed during that period, there can be race conditions, and as trading halt can occur for multiple reasons, and as trading venues need to ensure the book is in a valid state before resuming trading — they may need to adapt the book with latest updates before this.

How can I get a rebuilt order book from the incremental book?

BMLL has pre-rebuilt all limit order the L3 books that are available, and provide fields called AddOrderIndex and DelOrderIndex. Essentially these columns give the absolute positions at which orders need to be added or removed when processing updates (assuming books are lists).

FUNCTION replay_book(il3, output_predicate): // Initialize empty bid and ask books ask_book = [] bid_book = []

// Extract relevant columns doi = il3["DelOrderIndex"] aoi = il3["AddOrderIndex"] si = il3["Side"] // Side column (2 for ask, otherwise bid)

// Iterate over each row in il3 FOR i FROM 0 TO LENGTH(il3) - 1 DO: IF output_preficate(i, il3, bid_book, ask_book) THEN: YIELD bid_book, ask_book // Output the current state di = CONVERT_TO_INT(doi[i]) // Get delete index ai = CONVERT_TO_INT(aoi[i]) // Get add index

// Determine which book to modify IF si[i] == 2 THEN: book = ask_book ELSE: book = bid_book
// Delete order if applicable IF di >= 0 THEN: REMOVE_INDEX(book, di)
// Add order if applicable IF ai >= 0 THEN: INSERT_AT_INDEX(book, ai, il3[i])
IF output_condition(LENGTH(il3), il3, bid_book, ask_book) THEN: YIELD bid_book, ask_book